A True Positives Theorem for a Static Race Detector

NIKOS GOROGIANNIS, Facebook, UK and Middlesex University London, UK
PETER W. O'HEARN, Facebook, UK and University College London, UK
ILYA SERGEY*, Yale-NUS College, Singapore and National University of Singapore, Singapore

RacerD is a static race detector that has been proven to be effective in engineering practice: it has seen thousands of data races fixed by developers before reaching production, and has supported the migration of Facebook's Android app rendering infrastructure from a single-threaded to a multi-threaded architecture. We prove a True Positives Theorem stating that, under certain assumptions, an idealized theoretical version of the analysis *never reports a false positive*. We also provide an empirical evaluation of an implementation of this analysis, versus the original RacerD.

The theorem was motivated in the first case by the desire to understand the observation from production that RacerD was providing remarkably accurate signal to developers, and then the theorem guided further analyzer design decisions. Technically, our result can be seen as saying that the analysis computes an under-approximation of an over-approximation, which is the reverse of the more usual (over of under) situation in static analysis. Until now, static analyzers that are effective in practice but unsound have often been regarded as ad hoc; in contrast, we suggest that, in the future, theorems of this variety might be generally useful in understanding, justifying and designing effective static analyses for bug catching.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Concurrent programming structures**;

Additional Key Words and Phrases: Concurrency, Static Analysis, Race Freedom, Abstract Interpretation

ACM Reference Format:
Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL, Article 57 (January 2019), 29 pages. https://doi.org/10.1145/3290370

1 CONTEXT FOR THE TRUE POSITIVES THEOREM

The purpose of this paper is to state and prove a theorem that has come about by reacting to surprising properties we observed of a static program analysis that has been in production at Facebook for over a year.

The RacerD program analyzer searches for data races in Java programs, and it has had significantly more reported industrial impact than any other concurrency analysis that we are aware of. It was released as open source in October of 2017, and the OOPSLA'18 paper by Blackshear et al. (2018) describes its design, and gives more details about its deployment. They report, for example, that over 2,500 concurrent data races found by RacerD have been fixed by Facebook developers, and that it has been used to support the conversion of Facebook's Android app rendering infrastructure from a single-threaded to a multi-threaded architecture.

*Work done while employed as a part-time contractor at Facebook.

Authors' addresses: Nikos Gorogiannis, Facebook, UK, Middlesex University London, UK, nikosgorogiannis@fb.com; Peter W. O'Hearn, Facebook, UK, University College London, UK, peteroh@fb.com; Ilya Sergey, Yale-NUS College, Singapore, National University of Singapore, Singapore, ilya.sergey@yale-nus.edu.sg.

2475-1421/2019/1-ART57
https://doi.org/10.1145/3290370

RacerD's designers did not establish the formal properties of the analyzer, but it has been shown to be effective in practice. We wanted to understand this point from a theoretical point of view. RacerD is not sound in the sense of computing an over-approximation of some abstraction of executions. Over-approximations support a theorem: if the analyzer says there are no bugs, then there are none. *I.e.*, there are no false negatives (when the program has no bugs), which is often considered as a "soundness" theorem. RacerD favours reducing false positives over false negatives. A design goal was to "detect actionable races that developers find useful and respond to" but "no need to (provably) find them all". In fact, it is very easy to generate artificial false negatives, but Blackshear et al. (2018) say that few have been reported in over a year of RacerD in production.

One can react to this by saying that RacerD is simply an *ad hoc*, if effective, tool. But, the tool does not heuristically filter out bug reports in order to reduce false positives. It first does computations with an abstract domain, and then issues data race reports if any potential races are found according to the abstract domain. Its architecture is like that of a principled analyzer based on abstract interpretation, *even though it does not satisfy the usual soundness theorem.* This suggests that saying RacerD is ad hoc because it does not satisfy a standard soundness theorem is somehow missing something: It would be better if a demonstrably-effective analyzer with a principled design came with a theoretical explanation, even if partial, for its effectiveness. That is the research problem we set ourselves to solve in this work.

A natural question is if it is possible to actually modify RacerD so as to make it sound, without losing its effectiveness in generating *signal*—actionable and useful data race reports that developers are keen to fix. RacerD's initial design elided standard analysis techniques such as alias and escape analysis, on the grounds that this was consistent with the goal of reducing false positives. To try to get closer to soundness our colleague Sam Blackshear, one RacerD's authors, implemented an escape analysis to find race bugs due to locally declared references escaping their defining scope, *i.e.*, to reduce the false negatives. The escape analysis led to too many false positives; it contradicted the goal of high signal, and was not put into production. One of the current authors, Gorogiannis, tried another tack to reduce false negatives: a simple alias analysis, to find races between distinct syntactic expressions which may denote the same lvalue. Again the attempt caused too many false positives to make it to production.

Next we wondered: might there be a *different* theorem getting to the heart of why RacerD works? Because it is attempting to reduce false positives, a natural thing to try would be an *under-approximation* theorem. This would imply that every report is a true positive. This theorem is false for the analyzer, because it is possible to artificially generate false positives. One of the main reasons for this is that conditionals and loops are treated in a path-insensitive manner (*i.e.*, join corresponds to taking the union of potential racy accesses across the different branches).

It seems plausible to modify RacerD to be under-approximate in two ways, but each of these has practical problems. In the first way, one considers sets (disjunctions) of abstract states, and uses conjunction for interpreting `if` statements and disjunction at join points: this is like in symbolic execution for testing (Cadar and Sen 2013). This would cause scaling challenges because of the path explosion problem. RacerD runs quickly, in minutes, on (modifications to) millions of lines of code, and its speed is important for delivering timely signal. To make a much slower analysis would not be in the spirit of RacerD, and would not explain why the existing analyzer is effective. The other way would be to use a meet (like intersection) while sticking with one abstract state per program point. The problem here is that this prunes very many reports, so many that it would miss a great many bugs and (we reasoned) would not be worth deploying in Continuous Integration.

These considerations led us to the following hypothesis, which we would like to validate:

Conjecture (True Positives (TP) Theorem). *Under certain assumptions, the analyzer reports* no false positives.

An initial version of the assumptions was described as follows. Consider an idealized language, IL, in which programs have only *non-deterministic* choice in conditionals and loops, and where there no is recursion. Then the analyzer should only report true positives for that language.

The absence of booleans in conditionals in IL reflects an analyzer assumption that the code it will apply (well) to will use coarse-grained locking and not (much) fine-grained synchronization, where one would (say) race or not dependent on boolean conditions. In Facebook's Android code we do find fine-grained concurrency. The Litho concurrent UI library[1] has implementations of ownership transfer and double-checked locking, both of which rely on boolean conditions for concurrency control. This kind of code can lead to false positives for RacerD, but we did not regard that as a mistake in RacerD. For, the Android engineers advised us to concentrate on coarse grained locking, which is used in the vast majority of product code at Facebook. For instance, we rarely observed code calling into Litho which selects a lock conditionally based on the value of a mutable field. Thus, the TP theorem based on assumptions reflected in IL could, if true, be a way of explaining why the analyzer reports few false positives, even though it uses join for `if`-statements.

The no-recursion condition is there because we want to say that the analyzer gets the races right except for when divergence makes a piece of code impossible to reach. If a data race detector reports a bug on a memory access that comes after a divergent statement (`diverge; acc`) then this would be a false positive, but we would not blame data race reporting, but rather failure to recognize the divergence which makes the memory access unreachable. The no-recursion requirement is just a form of *separation of concerns* to help theory focus on explaining the data race reporting aspect. Note that if non-deterministic choices are the only booleans in `while`-loops, then such loops do not necessarily loop forever; that is why we forbid recursion and not loops here. We give a full description of the assumptions for the TP theorem in Section 3.

One can see our True Positives Theorem as establishing an *under-approximation* of an *over-approximation* in a suitably positioned *context*. Start with a program without recursion (context). Replace boolean values in conditionals and loop statements by non-determinism (over-approximation). Finally, report only true positives in that over-approximation (*i.e.*, under-approximate the potential bugs in that over-approximation). The more usual position in program analysis is to go for the reverse decomposition, an over-approximation of an under-approximation, to account for soundness relative to assumptions. The "under" comes about from ignored behaviors – *e.g.*, if an analyzer does not deal well with a particular feature, such as reflection, prune those behaviours involving reflection for stating the soundness property– and the "over" comes from a usual sound-for-bug-prevention analysis, but only *wrt.* this under-approximate model. In contrast, our notion of *under-of-over* seems like a good way to think about a static analysis for bug catching.

We explained above how we arrived at the statement of the TP Theorem by considering properties of a particular analyzer, paired with considerations of its effectiveness. We don't claim that to get an effective race detector you *must* end up in the position of the TP theorem. It might be possible to obtain a demonstrably useful-in-practice data race detector that satisfies a standard soundness (over-approximation) theorem. It might also be possible to find one satisfying a standard (unconditional) under-approximation theorem. In fact, there exist a number of dynamic data race detectors that are very promising, and at least one (Serebryany and Iskhodzhanov 2009) that is widely deployed in practice. Both of these directions are deserving of further research. The work here is not in conflict with these valuable research directions, but would simply complement them by providing new insights on designing *static* race analyzers.

[1] Available at https://fblitho.com.

Now, the True Positives Theorem was not actually true of RacerD when we formulated it, and apparently it is still not. But, subsequent to our discovery of the theorem, the authors of RacerD took it as a guiding principle. For example, they implemented a "deep ownership" assumption (Clarke and Drossopoulou 2002): if an access path, say `x.f`, is owned (*i.e.*, accessible through just one object instance), then so are all extensions, *e.g.* `x.f.g`. The analyzer would never report a race for an owned access. Somewhat surprisingly, even though the deep ownership assumption goes against soundness for bug prevention (over-approximation), it is compatible with the goal of reducing false positives and commonly holds in practice.

We have proven a version of the TP Theorem for RacerDX, a modified version of RacerD analyzer, which is based on IL and is not too far removed from the original. In this paper we formulate and prove the TP theorem for an idealized theoretical language, and we carry out an empirical evaluation of an implementation of RacerDX in relation to the in-production analyzer (RacerD), *wrt.* change in the the number of reports produced. The distance between RacerD and RacerDX is such that the latter, which has a precise theorem attached to it, makes on the order of 10-57% fewer data race reports on our evaluation suite. To switch RacerD for RacerDX in production would require confidence concerning amount of true bugs in their difference, or other factors (*e.g.*, simplicity of maintenance) which require engineering judgement; in general, replacing well-performing in-production software requires strong reasons. But, our experiments show how the basic design of RacerD is not far from an analyzer satisfying a precise theorem and, as we now explain, the extent of their difference is not so important for the broader significance of our results.

Broader Significance. Our starting point was a desire to understand theoretically why the specific analyzer RacerD is effective in practice, and we believe that our results go some way towards achieving this, but it appears that they have broader significance. They exemplify a way of studying program analysis tools designed for catching bugs rather than ensuring their absence.

The first point we emphasize is the following:

Unsound (and incomplete) static analyses can be principled, satisfying meaningful theorems that help to understand their behaviour and guide their design.

The concept 'sound' seems to have been taken as almost a synonym for 'principled' in some branches of the research community, and colleagues we have presented our results to have often reacted with surprise that such a theorem is possible. Representative are the remarks of one of the POPL referees: 'It is remarkable that the authors manage to prove any theorem at all about an unsound analysis' and the referees collectively who said 'proving a theorem for an unsound static analysis is a first'. Note that our our analysis is neither sound for showing the absence of bugs nor for bug finding (every report is a true positive: the static analysis often refers to this as 'complete', where in symbolic as well as concrete testing it is sometimes what is meant by 'sound'). The potential for theoretical insights on unsound and incomplete analyses is perhaps less widely appreciated than it could be. Such analyses are not necessarily *ad hoc* (although they can be).

Just to avoid misunderstanding, note that we are not talking about the common situation of where an analyzer is unsound generally but designed to be sound under assumptions. As described above, this situation can be understood as soundness (over-approximation) *wrt.* a different model than the usual concrete semantics, typically an under-approximation of the concrete semantics. By striving for minimizing false positives instead of false negatives, our analyzer is purposely designed to be unsound, and this difference is reflected in the *shape* of our theorem being under-of-over rather than the converse.

Much more significant than saying that *there exists* an unsound (and incomplete) analysis with a theorem, would be if we could make the above claim for an analyzer that was useful in the sense of helping people.

Here, the fact that RacerD (the analyzer deployed in production, without a theorem) and RacerDX (its close cousin, with a theorem) are not the same at first glance seems problematic. However, our experiments suggest strongly that, if RacerDX rather than RacerD had been put into production originally, it would have found thousands of bugs, far outstripping the (publicly reported) impact of all previous static race detectors. We infer:

One can have an unsound (and incomplete) but effective static analysis, which has significant industrial impact, and which is supported by a meaningful theorem; in our case the TP theorem.

The discussion of this second point is admittedly based on counter-factual reasoning (what if we 'had' deployed RacerDX instead of RacerD?), but the possibility of it being a false argument seems to be vanishingly small. The point is so powerful, and accepted by the Infer static analysis team members even outside the authors,[2] that the TP theorem is guiding the construction of new analysers at Facebook.

Thus, it seems that our results could be more broadly significant than the initial but perhaps worthwhile goal of 'understand why RacerD is effective'. The fact that RacerD and RacerDX don't coincide is not so important for the larger point, though it would have made for a neat story if they had. Such neat stories have so far been rather rare when developing analysers under industrial rather than scientific constraints. Future analysers, including ones being developed at Facebook, will possibly adhere more to the 'neat' story.

Paper Outline. In the remainder of the paper we give an overview of the intuition and reasoning principles for identifying and reporting concurrent data races in RacerD (Section 2). We then describe an idealized language (IL), whose set of features matches the common conventions followed in production Java code, as well as IL's over-approximating semantics (Section 3). We then provide a formal definition of RacerDX, a modified RacerD analysis, tailored for reporting no false positives, in the framework of abstract interpretation (Section 4). Our formal development culminates with a proof of the True Positives Theorem, coming in two parts: in Section 5 we prove completeness of RacerDX *wrt.* to its abstraction and in Section 6 we show how to reconstruct provably racy executions from the analysis results. In Section 7, we discuss the implementation of the theoretical analyzer described in Section 4. We then present an evaluation of RacerDX which compares it with RacerD on a set of real-world Java projects. We discuss related work in Section 8 and elaborate on the common formal guarantees considered for static concurrency analyses, positioning RacerDX amongst existing tools and approaches.

2 OVERVIEW

A textbook definition of a data race is somewhat low-level: a race is caused by potentially concurrent operations on a *shared memory location*, of which at least one is a write (Herlihy and Shavit 2008).

Data races in object-based languages with a language-provided synchronization mechanism (*e.g.*, Objective C or Java) can be described more conveniently, for the sake of being understood and fixed by the programmer, in terms of the program's *syntax* (rather than memory), via *access paths*, which would serve as runtime race "witnesses", by referring to the dynamic semantics of concurrent executions.[3] Given two programs (*e.g.*, calls to methods of the same instance of a Java class), C_1 and C_2, there is a *data race* between C_1 and C_2 if one can construct their concurrent execution trace τ, and identify two *access paths*, π_1 and π_2 (represented as field-dereferencing chains $x.\mathrm{f}_1.\ldots,\mathrm{f}_n$, where x is a variable or `this`), in C_1 and C_2, respectively, such that:

[2] RacerD and RacerDX are implemented using the Infer.AI abstract interpretation framework; see https://fbinfer.com.
[3] We will provide a more rigorous definition in Section 3.

(a) at some point of τ both π_1 and π_2 are involved into two concurrent operations, at least one of which is a write, while both π_1 and π_2 point to the same shared memory location;
(b) the sets of *locks* held by C_1 and C_2 at that execution point respectively, are disjoint.

This "definition" of a data race provides a lot of freedom for substantiating its components. Of paramount importance are (*i*) the considered set of pairs of programs that can race, (*ii*) the assumptions about the *initial state* of C_1 and C_2, and (*iii*) the notion of the *dynamic semantics*, employed for constructing concurrent execution traces. A choice of those determines what is considered to be a race and reflects some assumptions about program executions. For instance, accounting for the "worst possible configuration" (*e.g.*, arbitrary initial state and uncontrolled aliasing) would make the problem of sound (*i.e.*, over-approximating) reasoning about data races non-tractable in practice, or render its results too imprecise to be useful.

2.1 Race Detection in RacerD

As a specific set of design choices *wrt.* identifying data races in terms of access paths, let us consider RacerD (Blackshear et al. 2018)—a tool for static compositional race detection by Facebook—and its take on a toy example in Figure 1. The Java class `Dodo` has one **`private`** field `dee` and two **`public`** methods, both manipulating `dee`'s state. To instantiate the definition of a data race above, RacerD considers all pairs of public method calls of the *same* class instance as concurrently running programs C_1 and C_2 from the definition above (*i*). To instantiate the initial state, it assumes that method parameters of the same type (*e.g.*, `Dodo`) may alias, thus, maximizing a possibility of a race (*ii*). Finally, it assumes there is only one reentrant lock in the entire program (*e.g.*, `Dodo`'s **`this`**), and over-approximates the dynamic semantics by taking *all* branches of the conditionals and entering every loop (*iii*).

```
1  @ThreadSafe
2  public class Dodo {
3    private Dodo dee;
4
5    public void zap(Dodo d) {
6      synchronized (this) {
7        System.out.println(d.dee);
8      }
9    }
10   public void zup(Dodo d) {
11     d.dee = new Dodo();
12   }
13 }
```

Fig. 1. A racy Java class.

With this setup, RacerD reports the class `Dodo` (annotated with `javax.annotation.concurrent`'s `@ThreadSafe`) as racy, due to concurrent unsynchronized accesses to the field `dee` in two paths: a read from `d.dee` in the method `zap()` and the write to `d.dee` in `zup()`—both detected based on the assumption that the two `d`s can be run-time aliases, since they share the same type. This assumption makes it easy to reconstruct a concurrent execution of the two culprit methods, invoked with the same object, which exhibit a race, thus making this report a *true positive* of the analysis.

2.2 Fighting False Positives

By its nature, RacerD is a bug detector, hence it sacrifices the traditional concept of *soundness* (the property customary for static analyses stating that *all* behaviors of interest are detected (Cousot and Cousot 1979)) and, thus, may suffer *false negatives* (*i.e.*, miss races).[4] Instead, for the purposes of correctly detecting bugs, and minimizing the time programmers spend investigating the reports, RacerD emphasizes *completeness* (Ranzato 2013) over soundness, targeting what we are going to refer to as the *True Positives Conjecture*—that is, if it reports a data race, one should be able to find a witness execution and access paths exhibiting the concurrency bug, given the assumptions made.

As a next example, consider the Java class `Burble` in Figure 2. For the same reason as with `Dodo`, the methods `meps` and `reps` race with each other when run with aliased arguments. It is less obvious, however, whether `meps` races with `beps`—and in fact they do not! The reason for that is that `beps`

[4] Although there were very few reports of races missed in Facebook's production code (Blackshear et al. 2018).

```
1  class Bloop {
2    public int f = 1;
3  }
4
5  class Burble {
6
7    public void meps(Bloop b) {
8      synchronized (this) {
9        System.out.println(b.f);
10     }
11   }
12
13   public void reps(Bloop b) {
14     b.f = 42;
15   }
16
17   public void beps(Bloop b) {
18     b = new Bloop();
19     b.f = 239;
20   }
21 }
```

Fig. 2. A Java class with a false race.

```
22 class Wurble {
23   Wurble x = new Wurble();
24   Bloop g = new Bloop();
25
26   public void qwop(Wurble w) {
27     zwup(w.x);
28   }
29
30   public void gwap(Wurble w) {
31     synchronized (this) {
32       System.out.println(w.x.g);
33     }
34   }
35
36   private void zwup(Wurble w) {
37     synchronized (this) {
38       System.out.println(w.x.g);
39     }
40     w = new Wurble();
41     w.g.f = 21;
42   }
43 }
```

Fig. 3. A class with a false interprocedural race.

reassigns a freshly allocated instance of `Bloop` to the formal `b` before assigning to the field of the latter, thus, effectively *avoiding* a race with a concurrent access to `b.f` in `meps`.

This phenomenon of "destabilizing" an access path in a potentially racy program can be manifested both intra-procedurally (as in `Burble`) and inter-procedurally. To wit, in another example in Figure 3, the class `Wurble` demonstrates a similar instance of a *false race*, with the **`private`** method `zwup()` "destabilizing" the path `w.g` by assigning a newly allocated `Wurble` instance to `w`, thus, ensuring that `qwop()` and `gwap()` avoid a race with each other.

A sound static analyzer would typically be expected to report races in both of these examples, corresponding to a loss of precision. However, having a non-negligible number of *false positives* is not something a practical bug detector can afford.

To avoid this loss of signal effectiveness, RacerD employs an *ownership tracking domain* (Flanagan and Freund 2009; Naik et al. 2006), used to record the variables and paths that have been assigned a newly allocated object, thus remedying the situation shown above.

Upon closer examination, we found that RacerD's abstract domain, including that of ownership, was not enough to allow us to prove that an access path resolves to the same address, before and after execution. To wit, knowing that an access path `x.f.g` is *not* owned, does not guarantee that the lvalue it corresponds to stays the same during execution. The reason we wanted this latter property is that it is one of the simplest ways to exhibit a race: once we have set up an initial state where a path resolves to a certain address, and have shown that execution of $C_1 \mid\mid C_2$ does not modify that address (*i.e.*, the path to address is *stable*), we are in the position to uncoditionally say that if both programs access that address, they will race. The answer we came up with is that of *stability*; its negation, instability (or *wobbliness*), over-approximates ownership.

Thus we pose the question: can we state the reasonable (*i.e.*, non-trivial) conditions under which we can in confidence state (*i.e.*, formally prove) that *all* of RacerD's reported races are *true positives*?

We refer to this desirable result as the **True Positives Theorem** (TPT) for a static race detector, and in this paper we deliver such a theorem for a version of RacerD (called RacerDX, using stability), formulating a set of assumptions under which it holds, and assessing their practical implications and impact on signal.

2.3 A True Positives Theorem for RacerDX

Our main result enabling the TPT proof is defining an *over-approximating* concrete semantics and stating the sufficient conditions, both reflecting the behavior of production code, under which RacerDX, a modified version of RacerD, reports *no false data races*. The RacerDX True Positives Theorem, which substantiates this statement, builds on the following three pieces of the formal development that together form the central theoretical contribution of this work.

2.3.1 An Over-Approximating Concrete Semantics and Practical Assumptions. RacerDX is a flow- and path-insensitive static analysis, which goes in all branches of conditional expressions and loops. To account for this design choice while stating a formal completeness result, we adopt a novel non-deterministic (single-threaded) trace-collecting semantics that treats branch and loop guards as non-deterministically valued variables and explores all execution sequences (Section 3). We use this semantics to give meaning to single-threaded program executions, thus, building a *concrete domain* for the main RacerDX analysis procedure. Amongst other things, we assume just one global lock and a language with a single class (although we do not restrict the number of its fields and methods, as well as their signatures). We also restrict the reasoning to programs with *well-balanced locking* (*e.g.*, in Java terminology it would mean that only `synchronized`-enabled locking is allowed), and forbid recursion.

2.3.2 RacerDX *Abstraction and Sequential Completeness of its Analysis.* We formulate the abstract domain for RacerDX analysis (Section 4) along with the abstraction function to it from the concrete domain of the previously defined multi-threaded trace-collecting semantics, à la Brookes (2007), which we show to form a Galois connection (Cousot and Cousot 1979). We then prove, in Section 5, a tower of lemmas, establishing the "sequential" *completeness* of RacerDX's static analysis, which analyzes each sequential sub-program compositionally *in isolation*, without considering concurrent interleavings, with respect to this abstraction (Theorem 5.15). The novelty of our formal proof is in employing standard Abstract Interpretation, while taking a different perspective by establishing the completeness rather than soundness of an analysis in the spirit of the work by Ranzato (2013).

2.3.3 Syntactic Criteria for Ensuring True Positives. We introduce the notions of path *stability* and its counterpart, *wobbliness* (*i.e.*, instability) —simple syntactic properties, which are at the heart of stating the sufficient conditions for RacerDX's TPT— and connect it to the previously developed abstract domain and single-threaded RacerDX abstraction. A similar tower of lemmas is built, showing that a stable path resolves to the same address before and after execution, thus providing us with the tools for validating the reported race. Our formal development culminates in Section 6, with leveraging the sequential completeness result of RacerDX for *reconstructing* the concrete concurrent execution traces *exhibiting* provably true data races, thus delivering the final statement and the proof of TPT (Theorem 6.10).

2.4 Measuring the Impact of True Positives Theorem on Signal Effectiveness

The practical contribution of our work, described in Section 7, is an implementation of RacerDX, a revised version of RacerD, incorporating the analysis machinery enabling the result of TPT, and its evaluation. We aimed to measure the impact of employing stability as a sufficient condition for detecting true races with respect to the reduction in overall number of reported bugs.[5] We ran experiments contrasting RacerD and RacerDX on a number of open-source Java projects, ranging from 25k to 273k LOC. We evaluated the runtime of each analyser, and looked at the reports the two analyzers produced in several ways, in order to assess the loss of signal induced by RacerDX.

[5]It is easy to have a vacuous TPT by reporting no bugs whatsoever!

3 CONCRETE EXECUTION MODEL

To formally define our execution model and the notion of a race, we first describe an idealized programming language (IL) that accurately captures the essence of RacerDX's intermediate representation and faithfully represents the race-relevant aspects of Java and similar languages.

3.1 Language and Assumptions about Programs

We start by defining a programming language with a semantics suitable for RacerDX's goals. Our language is simplified compared to Java, with assumptions made to help our study of the question of whether the data race reports are effective.

A1 The language has only one class, that of record-like objects with an arbitrary, but finite set of fields (with names from a fixed set Field) which are themselves pointers.
A2 The concurrency model is restricted to exactly two threads, and use a single, reentrant lock. The commands `lock()` and `unlock()` are only allowed to appear in *balanced pairs* within block statements and method bodies.
A3 Local variables need no declaration, but must be assigned to before first use. Formal parameters are always taken from a fixed set. There are no global variables.
A4 There is no destruction of objects, only allocation.
A5 All methods are non-recursive, have call-by-value parameters and no return value. The assumption on non-recursion is standard, as we don't want to reason here about termination.
A6 Control for conditional statements and while loops is *non-deterministic*; there are no booleans.

Assumptions **A1** and **A5** are about ignoring potential sources of false positives which have nothing in particular to do with races. For example, if you enter an infinite loop before a potential data race then you would have a false positive if you reported the race, but we wouldn't expect a (static) race detector to detect infinite loops. Similarly, if you have a class that can't be inhabited you can't get races on it, but we view this as separate from the question of the effectiveness of the race reports themselves.

Assumption **A5** about recursion is perhaps not as practically restrictive as might first appear. Its impact is less than, for example, bounded symbolic model checking, which can be seen as performing a finite unwinding of a program to produce a non-recursive underapproximation of it (and without loops as well) before doing analysis. As we explained in Section 1, our main reason for making the assumption is a conceptual rather than a practical one, to do with separation of concerns, but we state the point about bounded model checking for additional context.

The reasons for the simplifications **A1** and **A2** are both related to RacerDX's focus on detection of races in one class at a time, with races manifested by parallel execution of methods on a single instance. Assumption **A2** is a potential source of real false negatives, as methods of the same class that use distinct locks can race. Furthermore, the well-balanced assumption in **A2** corresponds directly to scoped synchronisation mechanisms like Java's `synchronized(m){ }` construction (Goetz et al. 2006), where `m` is a static global mutex object. **A3** corresponds to RacerDX's intermediate language representation of parameters and variables. The lack of global variables is a genuine restriction, but which is easy to address; we discuss this from a practical point of view in Section 7. The assumption **A4** is due to the fact that Java is a garbage-collected language.

Finally, the assumption **A6** is the most significant one, both from the perspective of formalising a language semantics, and from the point of specifying completeness. By assuming that *every* execution branch may be taken, we do not have to reason statically about branching and looping conditions. This is effectively an *over-approximation* of the actual semantics of the concurrent programs. This assumption is motivated by two considerations: (i) RacerD purposely avoids tackling fine-grained concurrency, and is applied at Facebook to code where developers do not

$$
\begin{array}{rcl}
\mathsf{f} \in \mathsf{Field} & & \text{field names} \\
x, \mathtt{arg}_i \in \mathsf{Var} & & \text{variables} \\
\pi \in \mathsf{Path} & ::= & x.\mathsf{f} \mid \pi.\mathsf{f} \\
\mathsf{e} \in \mathsf{Exp} & \triangleq & \mathsf{Var} \cup \mathsf{Path} \\
c \in \mathsf{Stmt} & ::= & \mathtt{skip} \mid x := x \mid x := \pi \mid \pi := x \mid x := \mathtt{new}() \mid \mathtt{lock}() \mid \mathtt{unlock}() \mid \mathtt{pop}() \\
C \in \mathsf{CStmt} & ::= & c \mid C; c \mid C; \mathtt{if} * \mathtt{then}\ C\ \mathtt{else}\ C \mid C; \mathtt{while} * \mathtt{do}\ C \mid C; \mathsf{m}(\mathsf{e}, \ldots, \mathsf{e}) \\
M \in \mathsf{Method} & ::= & \mathsf{m}(\mathtt{arg}_1, \ldots, \mathtt{arg}_n)\ \{\ C\ \} \\
p \in \mathsf{Program} & ::= & C \parallel C
\end{array}
$$

Fig. 4. Syntax grammar of the concurrent programming language.

often avoid races by choosing which branch to take; (ii) RacerD uses join for **`if`** statements in order not to have too many false negatives.

The grammar defining the language of interest is given in Figure 4. We represent expressions $\mathsf{e} \in \mathsf{Exp}$ as either program variables and method formals $x \in \mathsf{Var}$, and access paths $\pi \in \mathsf{Path} = \mathsf{Var} \times \mathsf{Field}^+$. The language contains no constants such as, *e.g.*, **`null`**. We partition statements into *simple* and *composite*. Simple statements include assignments to variables, paths, reading from paths, allocating a new object, (blocking) lock acquisition via `lock()`, releasing a lock via `unlock()`, and popping an execution stack. The command `pop()` is not to be used directly in programs; it only occurs in our semantics of method calls and is needed for defining the trace-collecting semantics described below. Composite statements provide support for sequential composition (;), as well as conditionals ($\mathtt{if} * \mathtt{then} \cdot \mathtt{else} \cdot$), loops ($\mathtt{while} * \mathtt{do} \cdot$) and method calls, with the latter resulting in pushing a new stack frame on the call stack, as well as emitting a `pop()` command to remove it at runtime after the method body is fully executed.

Following the tradition of Featherweight Calculi for object-oriented languages, we introduce the function $\mathsf{mbody}(\cdot) : \mathsf{Method} \to \mathsf{CStmt}$, which maps method names to bodies and $\#(\cdot) : \mathsf{Method} \to \mathbb{N}$, which gives the number of formal parameters. We impose the convention that the formal parameters are always $\mathtt{arg}_1, \ldots \mathtt{arg}_n$ for all methods, where $n = \#(\mathsf{m})$ and that no other variables of the form $\mathtt{arg}_j$, where $j > n$, appear in $\mathsf{mbody}(\mathsf{m})$. We also allow local variables (*i.e.* variables x which are not of the form $\mathtt{arg}_i$) but forbid their use without definite prior initialisation.

3.2 Concrete Semantics

In this work, we do not tackle fine-grained concurrency (Turon et al. 2013), and we only consider lock-based synchronisation, as per assumption **A2**. Therefore, our concurrent semantics adopts the model of sequential consistency (Lamport 1979). To define it, we start by giving semantics to sequential program runs, which we will later combine to construct concurrent executions.

Our definition of runtime executions works over the following semantic categories. Loc denotes a countably infinite set of object *locations*. A stack[6] $s \in \mathsf{Stack}$ is a mapping $s : \mathsf{Var} \to \mathsf{Loc}$. Addresses Addr are defined in a field-splitting style, as $\mathsf{Addr} = \mathsf{Loc} \times \mathsf{Field}$. A heap $h \in \mathsf{Heap}$ is a partial, finite map $h : \mathsf{Addr} \to_{\mathrm{fin}} \mathsf{Loc}$. The constant nil is such that no heap is ever defined on an address of the form $(\mathsf{nil}, \mathsf{f})$. We write s_{nil} for the stack such that $s(x) = \mathsf{nil}$ for all $x \in \mathsf{Var}$.[7] The projection

[6] We overload the term *stack* to mean store here, borrowing from Separation Logic, the style of which informs most of our development. We disambiguate the term by explicitly using *call-stack* for a list of stacks.

[7] Here, nil is simply an unallocated address (corresponding to Java's `null`). We introduce it in order to avoid deviating too much from standard developments of shape analyses.

of an address to its first component is $\mathsf{locn}(\cdot)$, *i.e.*, $\mathsf{locn}(\ell, \mathrm{f}) = \ell$. We lift this definition to sets of addresses, $\mathsf{locn}(A) = \{\mathsf{locn}(\alpha) \mid \alpha \in A\}$, and to heaps, $\mathsf{locn}(h) = \mathsf{locn}(\mathrm{dom}\,(h))$. For example, $\mathsf{locn}(\{(\ell_1, \mathrm{f}) \mapsto \ell_2; (\ell_2, \mathrm{g}) \mapsto \ell_3\}) = \{\ell_1, \ell_2\}$. A lock context $L \in \mathsf{Locks} = \mathbb{N}$ is a natural number.

A (single-thread) *program state* $\varsigma \in \mathsf{State} = \mathsf{Stmt}^* \times \mathsf{Stack}^+ \times \mathsf{Heap} \times \mathsf{Locks}$ is a tuple $\langle c', S, h, L\rangle$, where $c' \in \mathsf{Stmt}^*$ is a either simple statement $c \in \mathsf{Stmt}$ (Figure 4) or a *runtime* call-statement $\mathtt{push}(e_1, \ldots, e_n)$; $S \in \mathsf{Stack}^+$ models the call-stack, h is a heap, and $L \in \mathsf{Locks}$. The stack at head position in the list S is the current stack frame. The L component is the lock context of the thread, signifying how many times a `lock()` instruction has been executed without a corresponding `unlock()`. We remark that the stack, heap and lock state components are those *produced by executing c starting at some previous program state.*

A *two-threaded* program state is a tuple $\left\langle c_{\|}, (S_1, S_2), h, (L_1, L_2)\right\rangle$, where $c_{\|}$ is either $c \parallel \epsilon$ or $\epsilon \parallel c$, denoting one of the two threads executing command c. The pairs (S_1, S_2) and (L_1, L_2) are the thread-local call-stacks and lock contexts for each thread, and h is the shared heap.

For the next series of definitions, we overload the term *state* to describe stack-heap pairs for both single-threaded and multi-threaded executions when the context is unambiguous.

Definition 3.1. The *address*, $\lfloor \pi \rfloor_{s,h}$, of a path π in a state (s, h) is recursively defined as follows:

$$\lfloor x.\mathrm{f} \rfloor_{s,h} \triangleq (s(x), \mathrm{f}) \qquad\qquad \lfloor \pi'.\mathrm{f} \rfloor_{s,h} \triangleq (h(\lfloor \pi' \rfloor_{s,h}), \mathrm{f})$$

The *value*, $[\![\mathrm{e}]\!]_{s,h}$, of an expression e in a state s, h is defined as follows:

$$[\![x]\!]_{s,h} \triangleq s(x) \qquad\qquad [\![\pi]\!]_{s,h} \triangleq h(\lfloor \pi \rfloor_{s,h})$$

The address of a path is the address read or written when a load or a store accesses that path.

Definition 3.2 (Execution trace). A (single-threaded) *execution* trace is a possibly empty list $\tau = [\varsigma_0, \ldots, \varsigma_n]$ of program states. The set of all traces is $\mathcal{T} = \mathsf{State}^*$.

We are now equipped with all the necessary formal components to give meaning to both single- and multi-threaded executions.

Definition 3.3 (Sequential trace-collecting semantics). A trace-collecting semantics of a single-threaded (possibly compound) program C, denoted $[\![C]\!] : \mathsf{Stack}^+ \times \mathsf{Heap} \times \mathsf{Locks} \to \wp(\mathcal{T})$ is a map to a set of traces starting from an initial configuration $\langle S, h, L\rangle$. The trace-collecting semantics are defined in Figure 5.[8] The auxiliary function $\mathsf{last}(\tau)$ (defined only on non-empty traces) returns the triple $\langle S', h', L'\rangle$ if the last element of the non-empty trace τ is $\langle c, S', h', L'\rangle$ for some command c.

We give the semantics for concurrent programs in the style of Brookes (2007).

Definition 3.4 (Concurrent execution trace). A (two-threaded) *concurrent execution* trace is a possibly empty list $\tau^{||} = [\varsigma_0^{||}, \ldots, \varsigma_n^{||}]$ of two-threaded program states. The set of all concurrent traces is $\mathcal{T}^{||}$.

Definition 3.5 (Concurrent trace-collecting semantics). The trace-collecting semantics of a parallel composition of programs C_1 and C_2 is defined in Figure 6. It is a map from two-threaded program states to sets of concurrent traces: $[\![C_1 \parallel C_2]\!] : (\mathsf{Stack}^+ \times \mathsf{Stack}^+) \times \mathsf{Heap} \times (\mathsf{Locks} \times \mathsf{Locks}) \to \wp(\mathcal{T}^{||})$. Intuitively, it interleaves all single-threaded executions of C_1 with those of C_2 taking care to guarantee that only one thread can hold the lock at a given step of the trace'.

We conclude this section of definitions by formally specifying concurrent data races.

[8] For the sake of uniformity, we assume that every program has the form $\mathtt{skip}; C$.

$$[\![\texttt{skip}]\!]\ \langle S,h,L\rangle \triangleq \{\epsilon\}$$

$$[\![x := \pi]\!]\ \langle s :: S,h,L\rangle \triangleq \begin{cases} \emptyset & \text{if } \lfloor\pi\rfloor_{s,h} \notin \mathrm{dom}\,(h) \\ \{[\langle x := \pi, s[x \mapsto h(\lfloor\pi\rfloor_{s,h})] :: S,h,L\rangle]\} & \text{otherwise} \end{cases}$$

$$[\![\pi := x]\!]\ \langle s :: S,h,L\rangle \triangleq \begin{cases} \emptyset & \text{if } \lfloor\pi\rfloor_{s,h} \notin \mathrm{dom}\,(h) \\ \{[\langle \pi := x, s :: S, h[\lfloor\pi\rfloor_{s,h} \mapsto s(x)], L\rangle]\} & \text{otherwise} \end{cases}$$

$$[\![x := \texttt{new}()]\!]\ \langle s :: S,h,L\rangle \triangleq \left\{ [\langle x := \texttt{new}(), s' :: S, h', L\rangle] \;\middle|\; \begin{array}{l} \ell \notin \mathrm{locn}(h), s' = s[x \mapsto \ell], \\ h' = h \cup \bigcup_{\mathsf{f}\in\mathsf{Field}} \{(\ell,\mathsf{f}) \mapsto \ell\} \end{array} \right\}$$

$$[\![x := y]\!]\ \langle s :: S,h,L\rangle \triangleq \{[\langle x := y, s[x \mapsto s(y)] :: S,h,L\rangle]\}$$

$$[\![\texttt{lock}()]\!]\ \langle S,h,L\rangle \triangleq \{[\langle \texttt{lock}(), S,h,\mathsf{add}(L,1)\rangle]\}$$

$$[\![\texttt{unlock}()]\!]\ \langle S,h,L\rangle \triangleq \begin{cases} \emptyset & \text{if } L \le 0 \\ \{[\langle \texttt{unlock}(), S,h,L-1\rangle]\} & \text{otherwise} \end{cases}$$

$$[\![\texttt{pop}()]\!]\ \langle s :: S,h,L\rangle \triangleq \{[\langle \texttt{pop}(), S,h,L\rangle]\}$$

$$[\![C;\mathsf{m}(\mathsf{e}_1,\ldots,\mathsf{e}_n)]\!]\ \langle S,h,L\rangle \triangleq \begin{cases} \emptyset & \text{if for some } i \le n, [\![\mathsf{e}_i]\!]_{s,h'} \text{ is undefined for} \\ & \langle s :: S',h',L'\rangle = \mathsf{last}([\![C]\!]\ \langle S,h,L\rangle) \\ \left\{ \tau \mathbin{+\!\!+} \tau_{\text{push}} \mathbin{+\!\!+} \tau' \;\middle|\; \begin{array}{l} \tau_{\text{push}} = [\langle \mathsf{push}(\mathsf{e}_1,\ldots,\mathsf{e}_n), \hat{s} :: s :: S',h',L'\rangle], \\ \tau \in [\![C]\!]\ \langle S,h,L\rangle, \langle s :: S',h',L'\rangle = \mathsf{last}(\tau) \\ \tau' \in [\![\mathsf{mbody}(\mathsf{m}); \mathsf{pop}()]\!]\ \langle \hat{s} :: s :: S',h',L'\rangle \\ \hat{s} = s_{\mathsf{nil}}[\mathtt{arg}_1 \mapsto [\![\mathsf{e}_1]\!]_{s,h}] \cdots [\mathtt{arg}_n \mapsto [\![\mathsf{e}_n]\!]_{s,h}] \end{array} \right\} \end{cases}$$

$$[\![C;c]\!]\ \langle S,h,L\rangle \triangleq \left\{ \tau \mathbin{+\!\!+} \tau' \;\middle|\; \begin{array}{l} \tau \in [\![C]\!]\ \langle S,h,L\rangle, \langle S',h',L'\rangle = \mathsf{last}(\tau), \\ \tau' \in [\![c]\!]\ \langle S',h',L'\rangle \end{array} \right\}$$

$$[\![C;(\texttt{if} * \texttt{then}\ C_1\ \texttt{else}\ C_2)]\!]\ \langle S,h,L\rangle \triangleq \left\{ \tau \mathbin{+\!\!+} \tau' \;\middle|\; \begin{array}{l} \tau \in [\![C]\!]\ \langle S,h,L\rangle, \langle S',h',L'\rangle = \mathsf{last}(\tau), \\ \tau' \in [\![C_1]\!]\ \langle S',h',L'\rangle \cup [\![C_2]\!]\ \langle S',h',L'\rangle \end{array} \right\}$$

$$[\![C;(\texttt{while} * \texttt{do}\ C')]\!]\ \langle S,h,L\rangle \triangleq \mathsf{fix}\,\mathcal{F}\ ([\![C]\!]\ \langle S,h,L\rangle), \text{ where}$$

$$\mathcal{F} \triangleq \lambda T.\ T \cup \left\{ \tau \mathbin{+\!\!+} \tau' \;\middle|\; \begin{array}{l} \tau \in T, \langle S',h',L'\rangle = \mathsf{last}(\tau), \\ \tau' \in [\![C']\!]\ \langle S',h',L'\rangle \end{array} \right\}$$

Fig. 5. Single-threaded trace-collecting semantics of simple (top) and compound statements (bottom).

Definition 3.6 (Data Race). The program $C_1 \parallel C_2$ *races* if there exists a state ς_0 and a non-empty concurrent trace $\tau \in [\![C_1 \parallel C_2]\!]\ \varsigma_0$ such that $\mathsf{last}(\tau) = \langle \_, (s_1 :: \_, s_2 :: \_), h, \_\rangle$[9] and,

- there exist paths π_1, π_2 such that $\lfloor\pi_1\rfloor_{s_1,h} = \lfloor\pi_2\rfloor_{s_2,h}$;
- there exist states $\varsigma_1 = \langle c_1 \parallel \epsilon, \_, \_, \_\rangle$ and $\varsigma_2 = \langle \epsilon \parallel c_2, \_, \_, \_\rangle$ such that $\tau :: \varsigma_1, \tau :: \varsigma_2 \in [\![C_1 \parallel C_2]\!]\ \varsigma_0$;
- $c_1 = (\pi_1 := \_) \wedge c_2 = (\pi_2 := \_)$, or, $c_1 = (\pi_1 := \_) \wedge c_2 = (\_ := \pi_2)$, or, $c_1 = (\_ := \pi_1) \wedge c_2 = (\pi_2 := \_)$.

How can this definition capture races without mentioning locks? For a thread to be blocked on a lock acquisition, the *successor* instruction to the current state *must* be a lock() statement, which is excluded by the syntactic condition on the successor instructions (*cf.* restrictions *wrt.*

[9] Throughout the paper we use the notation _ for we don't care about, effectively existentially quantifying them.

$$[\![C_1 \parallel C_2]\!]\ \langle (S_1,S_2),h,(L_1,L_2)\rangle \triangleq \bigcup_{\substack{\tau_1\in[\![C_1]\!]\ \langle S_1,h,L_1\rangle \\ \tau_2\in[\![C_2]\!]\ \langle S_2,h,L_2\rangle}} \mathsf{interl}\ \tau_1\ \tau_2\ \langle (S_1,S_2),h,(L_1,L_2)\rangle\,,\ \text{where}$$

$$\mathsf{interl}\ \epsilon\ \epsilon\ \langle (S_1,S_2),h,(L_1,L_2)\rangle \triangleq \{\epsilon\}$$

$$\mathsf{interl}\ \tau_1\ \tau_2\ \langle (S_1,S_2),h,(L_1,L_2)\rangle \triangleq \{\epsilon\} \cup$$

$$\left(\begin{array}{l} \{\langle C_1 \parallel \epsilon,(S_1,S_2),h,(L_1,L_2)\rangle \mid \tau_1 = \langle C_1,\_,\_,\_\rangle :: \_\} \cup \\ \{\langle \epsilon \parallel C_2,(S_1,S_2),h,(L_1,L_2)\rangle \mid \tau_2 = \langle C_2,\_,\_,\_\rangle :: \_\} \cup \\ \left\{ \langle C_1 \parallel \epsilon,(S_1,S_2),h,(L_1,L_2)\rangle :: \tau \;\middle|\; \begin{array}{l} \tau_1 = \langle C_1,\_,\_,\_\rangle :: \hat{\tau}_1 \wedge \\ \tau \in \mathsf{interl}\ \hat{\tau}_1\ \tau_2\ \left\langle (S_1',S_2),h',(L_1',L_2)\right\rangle \wedge \\ \left\langle C_1,S_1',h',L_1'\right\rangle \in [\![C_1]\!]\ \langle S_1,h,L_1\rangle \\ L_2 = 0 \vee L_1 = L_1' = 0 \end{array} \right\} \cup \\ \left\{ \langle \epsilon \parallel C_2,(S_1,S_2),h,(L_1,L_2)\rangle :: \tau \;\middle|\; \begin{array}{l} \tau_2 = \langle C_2,\_,\_,\_\rangle :: \hat{\tau}_2 \wedge \\ \tau \in \mathsf{interl}\ \tau_1\ \hat{\tau}_2\ \left\langle (S_1,S_2'),h',(L_1,L_2')\right\rangle \wedge \\ \left\langle C_2,S_2',h',L_2'\right\rangle \in [\![C_2]\!]\ \langle S_2,h,L_2\rangle \\ L_1 = 0 \vee L_2 = L_2' = 0 \end{array} \right\} \end{array}\right)$$

Fig. 6. Concurrent trace-collecting semantics.

locking contexts $L_i = L_i' = 0$ in Figure 6). Other sources of getting stuck are excluded by our assumptions **A2**–**A6**: there no deadlocks due to a single reentrant lock, no deterministic infinite loops and no recursion.

4 RACERDX ANALYSIS AND ITS ABSTRACT DOMAIN

The core RacerDX analysis statically collects information about path accesses occurring during an abstract program execution, as well as their locking contexts. In addition to those fairly standard bits, it tracks an additional program property, which makes it possible to filter out potential false negatives at the reporting phases—*wobbly* paths.

Definition 4.1. We call a path π *non-stable* (or *unstable*, or *wobbly*) in a program c if it appears as either LHS or RHS in a read or an assignment command during some execution of C. We elaborate this concept in the presence of method calls below.

Formally, the analysis operates on an abstract domain, which is a product of the three components: domain of *wobbly paths*, an *access path* domain and a *lock* domain.

Definition 4.2 (RacerDX abstract states). The abstract domain is $\mathcal{D} \triangleq \wp(\mathcal{W}) \times \mathcal{L} \times \mathcal{A}$, where

- $\mathcal{W} \triangleq \mathsf{Exp}$ is the set of wobbly accesses;
- $\mathcal{L} \triangleq \mathsf{Locks} = \mathbb{N}$ is the current lock context;
- $\mathcal{A} \triangleq \wp(\{\langle c, L\rangle \mid c \in \{\_ := \pi, \pi := \_\}, L \in \mathcal{L}\})$ is a domain of sets of recorded read/write accesses from/to paths which occur under lock state L.

We will use W, L, A as identifiers of elements of the corresponding domains above.

The lattice structure on $\mathcal{D}$ (which we will ofter refer to as *summaries*) is ordered (via $\sqsubseteq$) by pointwise lifting of the order relations $\langle \subseteq, \leq, \subseteq\rangle$ on the three components of the domain (Cousot and Cousot 1979). For an element $D \in \mathcal{D}$, we refer to the corresponding three projections as D_W, D_L and D_A.

$$\text{wobbly } W\ C = \begin{cases} W \cup \{x, \mathsf{e}\}^{\mathsf{fml}} & \text{if } C \equiv (x := \mathsf{e}) \text{ or } C \equiv (\mathsf{e} := x) \\ W \cup \{x\}^{\mathsf{fml}} & \text{if } C \equiv (x := \mathsf{new}()) \\ W & \text{for any other command } C \end{cases}$$

$$\text{locking } L\ C = \begin{cases} \mathsf{incr}(L) & \text{if } C \equiv (\mathtt{lock}()) \\ \mathsf{decr}(L) & \text{if } C \equiv (\mathtt{unlock}()) \\ L & \text{for any other command } C \end{cases}$$

$$\text{accs } (A, L)\ C = \begin{cases} A \cup \{\langle x := \pi, L\rangle\}^{\mathsf{fml}} & \text{if } C \equiv (x := \pi) \\ A \cup \{\langle \pi := x, L\rangle\}^{\mathsf{fml}} & \text{if } C \equiv (\pi := x) \\ A & \text{for any other command } C \end{cases}$$

$$\begin{aligned} [\![c]\!]^{\#}\langle W, L, A\rangle &= \langle \text{wobbly } W\ c, \text{locking } L\ c, \text{accs } (A, L)\ c\rangle \\ [\![C; c]\!]^{\#}\langle W, L, A\rangle &= [\![c]\!]^{\#}([\![C]\!]^{\#}\langle W, L, A\rangle) \\ [\![C; \mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)]\!]^{\#}\langle W, L, A\rangle &= [\![\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)]\!]^{\#}([\![C]\!]^{\#}\langle W, L, A\rangle) \\ [\![C; (\mathtt{if} * \mathtt{then}\ C_1\ \mathtt{else}\ C_2)]\!]^{\#}\langle W, L, A\rangle &= [\![C_1]\!]^{\#}([\![C]\!]^{\#}\langle W, L, A\rangle) \sqcup [\![C_2]\!]^{\#}([\![C]\!]^{\#}\langle W, L, A\rangle) \\ [\![C; (\mathtt{while} * \mathtt{do}\ C')]\!]^{\#}\langle W, L, A\rangle &= [\![C]\!]^{\#}\langle W, L, A\rangle \sqcup [\![C']\!]^{\#}([\![C]\!]^{\#}\langle W, L, A\rangle) \\ \text{where } \langle W_1, L_1, A_1\rangle \sqcup \langle W_2, L_2, A_2\rangle &= \langle W_1 \cup W_2, \max(L_1, L_2), A_1 \cup A_2\rangle \end{aligned}$$

$$\begin{aligned} [\![\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)]\!]^{\#}\langle W, L, A\rangle &= \langle W'', L'', A''\rangle \text{, where} \\ W'' &= W \cup \left\{\mathsf{e}_i \mid \exists j \neq i.\, \mathsf{e}_i \preccurlyeq \mathsf{e}_j\right\} \cup W'[\mathsf{e}_1/\mathtt{arg}_1, \ldots, \mathsf{e}_n/\mathtt{arg}_n], \\ L'' &= \mathsf{add}(L, L'), \\ A'' &= A \cup \{\langle c, \mathsf{add}(L, L_c)\rangle \mid \langle c, L_c\rangle \in A'\}\,[\mathsf{e}_1/\mathtt{arg}_1, \ldots, \mathsf{e}_n/\mathtt{arg}_n]^{\dagger} \\ \langle W', L', A'\rangle &= [\![\mathsf{mbody}(\mathsf{m})]\!]^{\#}\langle \emptyset, 0, \emptyset\rangle \end{aligned}$$

Fig. 7. Definition of the abstract analysis semantics $[\![\cdot]\!]^{\#} : \mathcal{D} \to \mathcal{D}$. We define $\mathsf{add}(L, L') = \min\{L + L', \hat{L}\}$, $\mathsf{incr}(L) = \mathsf{add}(L, 1)$, $\mathsf{decr}(L) = \max\{L - 1, 0\}$ and $\hat{L} \in \mathbb{N}^{+}$. †We define a substitution $\cdot[\theta]$ (applied recursively to all syntactic elements) as applying only to the path component of c (eg, $(\pi := x)[\theta] = (\pi[\theta] := x)$). The function fml acts as a filter that only selects expressions rooted at formals, *i.e.*, variables of the form $\mathtt{arg}_i$, and is extended straightforwardly to sets of elements of A, depending on the path component of the command.

For computing the join of branching control-flow and loops, the analysis employs a standard monotone version of a lub-like operator.

Definition 4.3 (Least-upper bound on $\mathcal{D}$). For $D_1 = \langle W_1, L_1, A_1\rangle$ and $D_2 = \langle W_2, L_2, A_2\rangle$,

$$D_1 \sqcup D_2 \triangleq \langle W_1 \cup W_2, \max(L_1, L_2), A_1 \cup A_2\rangle\,.$$

The intuition of what might go wrong with $\sqcup$ defined this way, when aiming for a "no-information-loss" analysis, is easier to see on an example. Consider the program $\mathtt{if} * \mathtt{then}\ C_1\ \mathtt{else}\ C_2; C$, where in C_1 the lock (which is reentrant, as agreed above) is taken strictly more times than in C_2; then by taking an over-approximation of the total number of times the lock is taken for both branches (*i.e.*, max), we have a chance to miss a race in the remainder program C, as some access path can be recorded by the analyzer as having a *larger* lock context than it would have in a concrete execution.

What comes to the rescue is the assumption **A2**, which we will exploit in our proofs of TP Theorem. Turns out, in practice, implementations of *all* classes we run the analysis on (*cf.* Section 7) have well-scoped locking, mostly relying on Java's `synchronized` primitive.

The definition of RacerDX analysis for arbitrary compound programs is given in Figure 7. The abstract transition function $[\![\cdot]\!]^\#$ relies on the three primitives, wobbly, locking, and accs, that account for the three corresponding components of the abstract domain. In the case of method calls, ($[\![\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)]\!]^\#$), the analysis also takes advantage of its own compositionality, adapting a summary of a method m to its caller context ($\langle W, L, A\rangle$) and the actual arguments $\mathsf{e}_1, \ldots, \mathsf{e}_n$.

What Does the Analysis Achieve? First, the results of the analysis from Figure 7 are used to construct a set of race candidates in RacerDX. Following the original RacerD algorithm described by Blackshear et al. (2018), if for a pair of method calls (possibly of the same method) of the same class instance, the same syntactic access path π appears in the both methods summaries, depending on the nature of the access (read or write) and the locking context L, at which the access has been captured, it might be deemed a race.

Second, the gathered wobbliness information comes into play. If a path π', which is a proper prefix of π, is identified as *wobbly* by the analysis, a race on π, *might* be a false positive. That is, wobbly paths "destabilise" the future results of the analysis, allowing the real path-underlying values "escape" the race, thus rendering it a false positive—precisely what we are aiming to avoid. Therefore, in order to report only true races, RacerDX removes from the final reports all paths that were affected by wobbly prefixes.

In Sections 5–6 we establish this completeness guarantee formally, showing that races reported on non-destabilised accesses are indeed *true positives*, in the sence that there *exists* a pair of execution traces for a pair of method calls of the same class instances that exhibit the behaviour described by Definition 3.6. Furthermore, the evaluation of RacerDX in Section 7 provides practical evidence that the notion of wobbliness *does not remove too* many reports, that is that the analysis remains efficient while being precise, in the assumptions **A1**–**A6**.

5 TOWARDS TP THEOREM, PART I: ANALYSIS COMPLETENESS

In this section, we deliver on the first part of the agenda towards True Positive Theorem: the definition of RacerDX *abstraction* (*wrt.* the concrete semantics), and the proof of *completeness* of the analysis with respect to it. That is, informally, if for a program C, the analysis $[\![C]\!]^\#$ reports a certain path π accessed under a locking context L, then *there exists* an initial configuration $\langle S, h, 0\rangle$ and an execution trace $\tau \in [\![C]\!]\,\langle S, h, 0\rangle$, such that τ contains a command accessing π with a locking context L. To establish this, we follow the general approach of the Abstract Interpretation framework (Cousot 1978; Cousot and Cousot 1977), aiming for completeness, *i.e.*, no information loss *wrt.* chosen abstraction (Ranzato 2013), stated in terms of a traditional abstract transition function and an abstraction from a concrete domain of sets of traces.

We first formulate the abstraction and prove its desirable properties. The remainder this arc of our story takes a "spiral" pattern, coming in two turns: establish the completeness of the analysis for straight-line programs (turn one), and then lift this proof for the inter-procedural case (turn two). In most of the cases, we provide only statements of the theorems, referring the reader to the extended version (Gorogiannis et al. 2018) for auxiliary definitions and proofs.

5.1 RacerDX Abstraction for Trace-Collecting Semantics

The abstraction connects the results of the analyzer to the elements of the concrete semantics, *i.e.*, traces. It is natural to restrict the considered traces and states to such that could indeed be produced by executions. We will refer to them as *well-formed* (WF) and *well-behaved* (WB).

$$
\begin{array}{lll}
\text{exec } \epsilon & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq (\langle W, L, A\rangle, \bar{\theta}') \\
\text{exec } \widehat{\tau} :: \langle x := y\rangle & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq \left(\left\langle W' \cup \{x, y\}^{\bar{\theta}'}, L', A'\right\rangle, \bar{\theta}'\right) \\
\text{exec } \widehat{\tau} :: \langle x := \pi\rangle & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq \left(\left\langle W' \cup \{x, \pi\}^{\bar{\theta}'}, L', A' \cup \{\langle x := \pi, L\rangle\}^{\bar{\theta}'}\right\rangle, \bar{\theta}'\right) \\
\text{exec } \widehat{\tau} :: \langle \pi := x\rangle & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq \left(\left\langle W' \cup \{x, \pi\}^{\bar{\theta}'}, L', A' \cup \{\langle \pi := x, L\rangle\}^{\bar{\theta}'}\right\rangle, \bar{\theta}'\right) \\
\text{exec } \widehat{\tau} :: \langle x := \mathsf{new}()\rangle & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq \left(\left\langle W' \cup \{x\}^{\bar{\theta}'}, L', A'\right\rangle, \bar{\theta}'\right) \\
\text{exec } \widehat{\tau} :: \langle \mathsf{lock}()\rangle & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq (\langle W', \mathsf{add}(L', 1), A'\rangle, \bar{\theta}') \\
\text{exec } \widehat{\tau} :: \langle \mathsf{unlock}()\rangle & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq (\langle W', \max(L' - 1, 0), A'\rangle, \bar{\theta}') \\
\text{exec } \widehat{\tau} :: \langle \mathsf{push}(\mathsf{e}_1, \ldots, \mathsf{e}_n)\rangle & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq \left(\begin{array}{l} \left\langle W' \cup \left\{\mathsf{e}_i \mid \exists j \neq i.\, \mathsf{e}_i \preccurlyeq \mathsf{e}_j\right\}^{\bar{\theta}'}, L', A'\right\rangle, \\ [\mathsf{arg}_i \mapsto \mathsf{e}_i]_{i=1}^{n} :: \bar{\theta}' \end{array}\right) \\
\text{exec } \widehat{\tau} :: \langle \mathsf{pop}()\rangle & (\langle W, L, A\rangle, \bar{\theta}) & \triangleq (\langle W', L', A'\rangle, \mathsf{tail}\ \bar{\theta}')
\end{array}
$$

$$\text{where } (\langle W', L', A'\rangle, \bar{\theta}') \triangleq \text{exec } \widehat{\tau}\ (\langle W, L, A\rangle, \bar{\theta}) \text{ and}$$

$$E^{\bar{\theta}} \triangleq \bigcup_{\mathsf{e} \in E} \mathsf{e}^{\bar{\theta}} \text{ for a set of expressions } E \qquad x.\bar{\mathsf{f}}^{\epsilon} \triangleq \begin{cases} \emptyset & \text{if } x \neq \mathsf{arg}_i \text{ for any } i \\ \{x.\bar{\mathsf{f}}\} & \text{otherwise} \end{cases}$$

$$x.\bar{\mathsf{f}}^{\theta :: \bar{\theta}} \triangleq \begin{cases} \emptyset & \text{if } x \notin \mathrm{dom}(\theta) \\ ((\mathsf{arg}_i[\theta]).\bar{\mathsf{f}})^{\bar{\theta}} & \text{if } x = \mathsf{arg}_i \in \mathrm{dom}(\theta) \end{cases} \qquad \{x := \pi\}^{\bar{\theta}} \triangleq \begin{cases} \emptyset & \text{if } \pi^{\bar{\theta}} = \emptyset \\ \{x := \pi'\} & \text{if } \pi^{\bar{\theta}} = \{\pi'\} \end{cases}$$

Fig. 8. An auxiliary function exec for executing syntactic traces to compute RacerDX abstraction.

Definition 5.1. An execution trace is well-formed if for each two subsequent states ς_i and ς_{i+1}, the stack and heap of ς_{i+1} can be obtained by executing the simple command in ς_{i+1}'s first component with respect to the stack/heap of ς_i.

It is easy to show that for any program C, S, h, L trace $\tau \in [\![C]\!]\ \langle S, h, L\rangle$, τ is well-formed.

Definition 5.2. A program state $\langle \_, (S_1, S_2), h, \_\rangle$ is well-behaved *iff* (a) for any s appearing in S_1, S_2 and any variable x, $s(x) \in \mathrm{locn}(h)$, and, (b) for any address $\alpha \in \mathrm{dom}(h)$, $h(\alpha) \in \mathrm{locn}(h)$.

We remark that dangling pointers do not occur in Java, and we reproduce a similar result for our language below.

Lemma 5.3 (Preservation of well-behavedness). *Let τ be a non-empty, well-formed trace, whose starting state is well-behaved. Then, every state in τ is well-behaved.*

Abstract Domain and Abstraction Function. The analysis's domain (Definition 4.2) is rather coarse-grained: it does not feature any information about runtime heaps or stacks. To bring the concrete traces closer to abstract summaries, for an execution trace τ, we define *syntactic trace* $\widehat{\tau}$ as a list, obtained by taking only the first components of each of τ's elements. That is, $\widehat{\tau} \triangleq \mathrm{map}\ (\lambda\ \langle C, -, -, -\rangle .C)\ \tau$. We now define the abstraction function $\alpha : \wp(\mathcal{T}) \to \mathcal{D}$, from the lattice of sets of execution traces $\wp(\mathcal{T})$ to $\mathcal{D}$, which corresponds to "folding" a syntactic trace, encoding a run of straight-line program (equivalent to the original program for this particular execution), left-to-right, to an abstract state recursively.

Definition 5.4 (Abstraction Function). For a set of well-formed traces $T \subseteq \mathcal{T}$,

$$\alpha(T) = \bigsqcup_{\tau \in T} (\text{fst } (\text{exec } \widehat{\tau}\ (\bot_{\mathcal{D}}, \epsilon))), \text{ where } \bot_{\mathcal{D}} = \langle \bot_{\mathcal{W}}, \bot_{\mathcal{L}}, \bot_{\mathcal{A}} \rangle$$

and exec : $\mathcal{T} \to \mathcal{D} \to \mathcal{D} \times (\text{Var} \rightharpoonup_{\text{fin}} \text{Exp})$ is defined in Figure 8.[10]

The $\bar{\theta}$ component in the state carried forward by exec is a stack of substitutions (from formals, $\texttt{arg}_i$, to paths) which mirrors the call stack in a concrete execution. Whenever an access occurs, the substitutions $\bar{\theta}$ are immediately applied, and the result is a path that is rooted at a top-level variable $\texttt{arg}_i$. Accesses rooted at local variables (any variable x which is not in the syntactic form $\texttt{arg}_i$ or not in the domain of the top-most substitution) are discarded (the substitution function returns an empty set). Similarly, the set W is populated with expressions that have the substitutions $\bar{\theta}$ already applied, and similarly discarded if rooted at a local variable.

The commands push() and pop() accordingly manipulate the substitution stack, while push() also adds certain extra paths to W; this is to avoid the effects of aliasing of paths rooted at different formals inside the method body. That is, paths can become wobbly because parts of the same path have been provided as parameters in a method call, such as in $\mathsf{m}(x.\mathsf{f}, x.\mathsf{f}.\mathsf{g})$. The full reasoning behind this definition will become clearer in Section 6, where it is used in the construction of a memory state where there is exactly one parameter pointing to the heap-image of the racy path.

We next establish a number of facts about α necessary for the proof of our analysis completeness.

Lemma 5.5 (Additivity of α). *α is additive (*i.e.*, preserves lubs) with respect to $\cup_{\wp(\mathcal{T})}$ and $\sqcup_{\mathcal{D}}$.*

Thanks to Lemma 5.5, we can define the (monotone) Galois connection $\langle \wp(\mathcal{T}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}, \sqsubseteq \rangle$ between the two complete lattices, where $\gamma \triangleq \lambda a. \bigcup \{T \in \wp(\mathcal{T}) \mid \alpha(T) \sqsubseteq a\}$ (Cousot and Cousot 1992). Having a Galois connection between $\wp(\mathcal{T})$ and $\mathcal{D}$ in conjunction with completeness of the analysis (Theorem 5.15) allows us to argue for the presence of a certain accesses in some concrete trace of a program C, if the analysis reports it for C. This is due to the following fact establishing that if α records an access in a certain trace, then such a path was indeed present in its argument:

Lemma 5.6 (Path access existence). *Let T be a set of traces, $\alpha(T) = \langle \_, \_, A \rangle$, and $q = \langle c, L \rangle$ (where $c = (x := \pi)$ or $c = (\pi := x)$) is a query about the access path π in the locking context L. If $q \in A$ then there exist a trace $\tau \in T$ and a non-empty, shortest prefix $\tau' \preccurlyeq \tau$ such that*

- *the last state of τ' is $\langle c', \_, \_, L \rangle$ and c', c are both stores or loads;*
- *exec $\widehat{\tau'}\ (\bot_{\mathcal{D}}, \epsilon) = (\langle \_, \_, A \rangle, \bar{\theta})$ where $\pi \in A$;*
- *a path π' such that $c' = (x := \pi')$ or $c' = (\pi' := x)$, and $\{\pi\} = \{\pi'\}^{\bar{\theta}}$.*

Proof. By the definition of $\alpha(\cdot)$ and the properties of $\bigsqcup$ follows that there must exist a trace $\tau \in T$ such that $q \in \alpha(\{\tau\})_A$. By the definition of exec the other elements follow directly. □

5.2 Proving that the Analysis Loses No Information

We structure the proof in two stages: first considering only straight-line programs with no method calls (Section 5.2.1) and then lifting it to programs with finite method call hierarchies (Section 5.2.2). We do not consider the cases with recursive calls (*cf.* Assumption **A5**).

5.2.1 Intra-Procedural Case. The abstract transfer function of the analysis for simple commands is defined in Figure 7 as $[\![c]\!]^{\#}\langle W, L, A \rangle$. We first prove the completeness for simple commands for a singleton-trace concrete domain.

[10] We overload the list notation $\widehat{\tau} :: c$ to denote appending a list $[c]$ to $\widehat{\tau}$, *i.e.*, $\widehat{\tau} :: c \triangleq \widehat{\tau}\ +\!\!+ [c]$.

Lemma 5.7 (Analysis is complete for simple commands (per-trace)). *For any non-empty WF trace $\tau \in \mathcal{T}$, sets W, A, number L, and a simple command c, which is not* pop(), *such that (a) $\langle W, L, A\rangle = \alpha(\{\tau\})$, (b) $[\![c]\!]\ \mathsf{last}(\tau) = \{[\varsigma] \mid \varsigma$ is an execution state$\}$, where $\mathsf{last}(\tau)$ is the configuration $\langle S, h, L\rangle$ of the last element of τ, the following holds:*

$$[\![c]\!]^{\#}\langle W, L, A\rangle = \alpha\left(\bigcup \left\{\tau :: \varsigma \mid [\varsigma] \in [\![c]\!]\ \mathsf{last}(\tau)\right\}\right).$$

The following result lifts the reasoning of Lemma 5.7 from singletons to sets of arbitrary traces.

Lemma 5.8 (Analysis of simple commands is complete for sets of traces). *For any set T of non-empty well-formed traces, W, L, A, and simple command c, which is not* pop(), *such that (a) $\langle W, L, A\rangle = \alpha(T)$, (b) for any $\tau \in T$, $[\![c]\!]\ \mathsf{last}(\tau) = \{[\varsigma] \mid \varsigma$ is an execution state$\}$, then*

$$[\![c]\!]^{\#}\langle W, L, A\rangle = \alpha\left(\bigcup \left\{\tau :: \varsigma \mid \tau \in T, [\varsigma] \in [\![c]\!]\ \mathsf{last}(\tau)\right\}\right).$$

The fact of preserving the equality of abstract results in Lemmas 5.7 and 5.8 is quite noteworthy: for straight-line programs (and, in fact, for any program in our IL) the analysis is *precise*, *i.e.*, we do not lose information *wrt.* locking context, wobbliness, or access paths, and hence can include elements $\langle c, L\rangle$ into the A-component as a part of accs-machinery without the loss of precision.

We now lift these facts to the analysis for compound programs without method calls. Recall that we only consider programs with *balanced locking* (Assumption **A2**), *i.e.*, such that within them

- the commands `lock()` and `unlock()` are only allowed to appear in balanced pairs (including their appearances in nested method calls) within conditional branches and looping statements, and,
- every `unlock()` command has a matching `lock()`.

The following two lemmas are going exploit this fact for proving the analysis completeness.

Lemma 5.9 (RacerDX analysis and balanced locking). *If C is a compound program with balanced locking, and $\langle W', L', A'\rangle = [\![C]\!]^{\#}\ \langle W, L, A\rangle$. Then $L' = L$.*

Lemma 5.10 (Balanced locking and syntactic traces). *For any program C with balanced locking and well-behaved states S_1, h_1 and S_2, h_2 and $L \geq 0$,*

$$\{\hat{\tau} \mid \tau \in [\![C]\!]\ \langle S_1, h_1, L\rangle)\} = \{\hat{\tau} \mid \tau \in [\![C]\!]\ \langle S_2, h_2, L\rangle)\} \neq \emptyset.$$

As a corollary, $\alpha([\![C]\!]\ \langle S_1, h_1, 0\rangle) = \alpha([\![C]\!]\ \langle S_2, h_2, 0\rangle)$.

The proof of Lemma 5.10 hinges on the the following observations:

- Balanced locking ensures that $[\![\texttt{unlock()}]\!]\ \langle S, h, L\rangle \neq \emptyset$, which the case for any intermediate state of the traces of the programs in consideration.
- Well-behavedness ensures that the set of traces for loads and stores is non-empty.
- Well-behavedness is preserved along traces (*cf.* the semantics of new()).

That is, for compound programs with balanced lockings the L-component of the abstraction remains immutable (by the end of execution) for both $[\![\cdot]\!]^{\#}$ and the abstraction α over the concrete semantics. The following lemma delivers the completeness in the intra-procedural case.

Lemma 5.11. *For any compound program C with balanced locking and no method calls, the starting components W_0, L_0, A_0, and a set of well-formed non-empty traces T, such that $\langle W_0, L_0, A_0\rangle = \alpha(T)$,*

$$[\![C]\!]^{\#}\ \langle W_0, L_0, A_0\rangle = \begin{cases} \alpha\left(\bigcup_{\tau \in T} \left\{\tau +\!\!+ \tau' \mid \tau' \in [\![C]\!]\ \mathsf{last}(\tau)\right\}\right) & \text{if } T \neq \emptyset, \\ \alpha\left([\![C]\!]\ \langle S, h, 0\rangle\right) & \text{otherwise, for any well-behaved } S, h, \end{cases}$$

where $\mathsf{last}(\tau)$ is well-defined, as T consists of non-empty traces.

Proof. By induction on C and Lemma 5.8. Detailed proof is in (Gorogiannis et al. 2018). □

5.2.2 Inter-Procedural Case. The main technical hurdle on the way for extending the statement of Lemma 5.8 to an inter-procedural case (*i.e.*, allowing for method calls) is the gap between the semantic implementation of method calls via stack-management discipline (Figure 5) and treatment of method summaries by the analysis, via explicit substitutions (Figure 7). To address this, we introduce the following convention, enforced by RacerDX's intermediate language representation:

Definition 5.12 (ANF). We say that a method $\mathsf{m}(\mathsf{arg}_1, \ldots, \mathsf{arg}_n)$ is in Argument-Normal Form (ANF), if for every simple command c in its body, c is *not* of the form $\pi := \mathsf{arg}_j$ for some $\mathsf{arg}_j \in \overline{\mathsf{arg}_i}$.

Intuitively, this requirement enforces a "sanitisation" of used arguments from the set $\overline{\mathsf{arg}_i}$, so one could *substitute* them with some (non-variable) paths without disrupting the syntactic structure of a compound program. Let us denote as $\mathrm{WB}(\mathsf{e}_{1\ldots n})$ the set of paths $\left\{\mathsf{e}_i \mid \exists j \neq i.\, \mathsf{e}_i \preccurlyeq \mathsf{e}_j \vee \mathsf{e}_j \preccurlyeq \mathsf{e}_i\right\}$ for $i \in \{1 \ldots n\}$. We will also denote via $\cup_1$ the following operation on $\mathcal{D}$:

$$\langle W, L, A\rangle \cup_1 W' = \langle W \cup W', L, A\rangle\,.$$

The following lemma accounts for the mentioned concrete/abstract discrepancy in treatment of methods, allowing us to reformulate the definition of abstract method summaries returned by the analyzer ($[\![\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)]\!]^\# \langle W, L, A\rangle$) via substitutions of their bodies.

Lemma 5.13 (ANF and method summaries). *For a method* m*, a vector of expressions* $e_1, \ldots, e_n$*, a configuration* $\langle s :: S, h, L\rangle = \mathsf{last}(\tau)$ *such that*

- $\forall i, 1 \leq i \leq n, [\![e_i]\!]_{s,h}$ *are defined, and*
- mbody(m) *has no nested calls and features well-balanced locking, the following holds:*

$$[\![\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)]\!]^\# \langle W, L, A\rangle \;=\; \langle W', L', A'\rangle \cup_1 \mathrm{WB}(\mathsf{e}_{1\ldots n}),\ \textit{where}$$
$$\langle W', L', A'\rangle = [\![\mathsf{mbody}(\mathsf{m})[\overline{\mathsf{e}_i/\mathsf{arg}_i}]]\!]^\# \langle W, L, A\rangle$$

The first Good Thing afforded by Lemma 5.13 is the removal of substitutions from the W and A components and moving them to the method's body, which will give the uniformity necessary for conducting the forthcoming proofs. The second Good Thing is the given possibility to "shift" the computation of the method summary from the $\bot_{\mathcal{D}} = \langle \emptyset, 0, \emptyset\rangle$ (as given by Figure 7) to the actual abstract context $\langle W, L, A\rangle$. Finally, notice that in the absence of recursion, the statement of Lemma 5.13 can be generalised to methods with nested calls, with the proof by induction on the size of the call tree.

Armed by Lemma 5.13 (generalised to nested calls), we can prove analogues of Lemmas 5.7–5.11. In the presence of method calls. The formal development first establishes the result for a singleton trace, similar to Lemma 5.7 for a program with no *nested* method calls, then lifting it, by induction on the size of the call graph, to arbitrary non-recursive call hierarchies. It is then further generalised for sets of traces, in a way similar to the proof of Lemma 5.7. For the sake of saving space, we do not present the statements of this lemmas here (as they are not particularly remarkable or pretty) and refer the reader the extended version (Gorogiannis et al. 2018) for auxiliary definitions and proofs. This arc of formal results concludes with the following statement, similar to Lemma 5.11:

Lemma 5.14. *For any compound program* C *with balanced locking and all methods in ANF, the analysis domain components* W_0, L_0, A_0, *and a set of well-formed non-empty traces* T, *such that* $\langle W_0, L_0, A_0\rangle = \alpha(T)$,

$$[\![C]\!]^\# \langle W_0, L_0, A_0\rangle = \begin{cases} \alpha\left(\bigcup_{\tau \in T} \left\{\tau +\!\!+ \tau' \mid \tau' \in [\![C]\!]\ \mathsf{last}(\tau)\right\}\right) & \textit{if } T \neq \emptyset, \\ \alpha\left([\![C]\!]\ \langle S, h, 0\rangle\right) & \textit{otherwise, for any well-behaved } S, h, \end{cases}$$

where $\mathsf{last}(\tau)$ *is well-defined, as* T *consists of non-empty traces.*

5.2.3 Main Completeness Result. We are now ready to establish the completeness of the abstract semantics $[\![\cdot]\!]^\#$ of our analysis *wrt.* trace-collecting semantics $[\![\cdot]\!]$ and abstraction α.

Theorem 5.15 (Completeness of RacerDX analysis.). *For any compound program C with all methods in ANF, balanced locking, and well-behaved S, h,*

$$[\![C]\!]^\# \bot_{\mathcal{D}} = \alpha\left([\![C]\!]\langle S, h, 0\rangle\right).$$

Proof. Follows immediately from Lemma 5.14 by taking $\langle W_0, L_0, A_0\rangle = \bot_{\mathcal{D}}$, $T = \emptyset$. □

6 TOWARDS TP THEOREM, PART II: RECONSTRUCTING RACY TRACES

Here we show how, given a pair of programs C_1, C_2 for which RacerDX reports a race, we *construct* a concurrent trace (as defined in Figure 6) that *exhibits the race*, in the sense of Definition 3.6.

We crucially rely on the notion of *stability* (*non-wobblyness*): an access path π is stable if no proper prefix of π is ever read or written. We track the negation of this property through the W component of the abstract state, which is slightly complicated by our need to track accesses to proper prefixes of π even inside procedure calls, and thus employ substitutions of actuals over formals (cf. the definition of exec for load/stores and calls). Note that stability does not preclude accesses to the path itself. Stability allows us to prove that the backbone of a path (the domain of its heap-image) is preserved during execution, and thus when the path is accessed at the end of a trace for C_1/C_2, the same address in both threads is accessed.

However, not all traces of C_1/C_2 will allow us to do this. For this reason we introduce the notions of *path disconnectedness* and *acyclicity*, two semantic restrictions that ensure that accesses through paths syntactically distinct to π, and suffixes of π respectively, do not affect the heap-image of π.

Definition 6.1 (Prefix). An access path $\pi = x.\bar{\mathrm{f}}$ is a prefix of $\pi' = y.\bar{\mathrm{g}}$ if $x = y$ and the sequence fs is a prefix of gs. We denote this fact by $\pi \preccurlyeq \pi'$. The notion of proper prefix ($\prec$) is straightforward. We lift this to expressions by fixing $x \prec x.\bar{\mathrm{f}}$ and $x \preccurlyeq x$.

The root of an access path $x.\mathrm{fs}$ is x, *i.e.*, $\mathrm{root}(x.\bar{\mathrm{f}}) = x$. If $\pi = x.\bar{\mathrm{f}}$, we may write $\pi.\mathrm{g}$ or $x.\bar{\mathrm{f}}.\mathrm{g}$ to mean $x.(\bar{\mathrm{f}}.\mathrm{g})$. We set $|x.\bar{\mathrm{f}}| = |\bar{\mathrm{f}}|$, *i.e.*, the length of the field list.

Definition 6.2 (Path Footprint). Let s, h be a state where $\lfloor\pi\rfloor_{s,h}$ is defined. Define the *footprint of* π, denoted $h{\downarrow}^s_\pi$, as

$$h{\downarrow}^s_\pi = \bigcup_{\pi' \preccurlyeq \pi} \{\lfloor\pi'\rfloor_{s,h} \mapsto h(\lfloor\pi'\rfloor_{s,h})\}.$$

Intuitively, the footprint of an access path is a heap fragment containing all addresses necessary to resolve the address of the path, inclusive. It's easy to see that

$$\mathrm{dom}\left(h{\downarrow}^s_\pi\right) = \{\lfloor\pi'\rfloor_{s,h} \mid \pi' \preccurlyeq \pi\}.$$

Definition 6.3 (Disconnectedness). A path π is *disconnected in a state s, h* if $\lfloor\pi\rfloor_{s,h}$ is defined, and

- for any $x \neq \mathrm{root}(\pi)$, $s(x) \notin \mathrm{locn}(h{\downarrow}^s_\pi)$;
- for all $\alpha \in \mathrm{dom}\,(h)$, if $h(\alpha) \in \mathrm{locn}(h{\downarrow}^s_\pi)$ then $\alpha \in \mathrm{dom}\left(h{\downarrow}^s_\pi\right)$.

Definition 6.4 (Acyclicity). A path π is *acyclic* in state s, h if $\lfloor\pi\rfloor_{s,h}$ is defined, and for any two paths $\pi'' \preccurlyeq \pi' \preccurlyeq \pi$, $h(\lfloor\pi'\rfloor_{s,h}) \neq \mathrm{locn}(\lfloor\pi''\rfloor_{s,h})$.

Definition 6.5 (Path preservation). Let π be a path, and s, h and s', h' two states. We say that π is preserved from s, h to s', h' if

(1) π is disconnected and acyclic in s, h and s', h';
(2) $s(\mathrm{root}(\pi)) = s'(\mathrm{root}(\pi))$;

(3) $\mathrm{dom}\left(h{\downarrow}^{s}_{\pi}\right) = \mathrm{dom}\left(h'{\downarrow}^{s'}_{\pi}\right)$;
(4) $\forall \alpha \in \mathrm{dom}\left(h{\downarrow}^{s}_{\pi}\right) \setminus \{\lfloor \pi \rfloor_{s,h}\}. h(\alpha) = h'(\alpha)$.

It is easy to see that if π is preserved from s, h to s', h', then $\lfloor \pi \rfloor_{s,h} = \lfloor \pi \rfloor_{s',h'}$. Also, is is straightforward to see that path preservation is an equivalence relation on memory states.

We begin by showing that a trace without calls, for which π is stable, preserves π.

Lemma 6.6. *Let π be an access path and τ a WF, non-empty trace without any* push()/pop() *commands, whose starting state $\langle \mathtt{skip}, s_0 :: S, h_0, \_\rangle$ is well-behaved, and where π is acyclic and disconnected in s_0, h_0. Let the last state of τ be $\langle \_, s' :: S, h', \_\rangle$.*
If it is the case that $\forall \mathsf{e} \in \alpha(\{\tau\})_W.\mathsf{e} \not\prec \pi$, then π is preserved from s_0, h_0 to s', h'.

Proof. By induction on the length of τ and results on stability. See the extended version (Gorogiannis et al. 2018) for details of the proof. □

We can now lift this up to the trace of a single, non-nested procedure call.

Lemma 6.7. *Suppose that* mbody(m) *has no procedure calls, that π is an access path which is disconnected and acyclic in the well-behaved state s, h, that $\tau \in [\![\mathtt{skip}; \mathsf{m}(\bar{\mathsf{e}})]\!] \langle s, h, L\rangle$ is a non-empty trace, that* last$(\tau) = \langle \mathsf{pop}(), s, h', _\rangle$*, that $\alpha(\{\tau\}) = \langle W, \_, \_\rangle$, and that $\forall \mathsf{e} \in W.\, \mathsf{e} \not\prec \pi$.*
Then, π is preserved from s, h to s, h'.

Proof. First, notice that $\forall \mathsf{e} \in W.\, \mathsf{e} \not\prec \pi$ and the definition of exec implies there is at most one e_i such that $\mathsf{e}_i = z$, or $\mathsf{e}_i = \pi_i \prec \pi$. We then analyse the three cases separately: no $\mathsf{e}_i \prec \pi$, $\mathsf{e}_i = \mathrm{root}(\pi)$ and $\mathsf{e}_i \prec \pi$. The most interesting is the last one, where we cut out the subheap of h reachable from the arguments, apply Lemma 6.6 to it, and then restore the facts we prove back to the top-level stack. Notably, here we use the *frame property* at the semantics level (Yang and O'Hearn 2002). □

We can further lift the preservation result to traces that have arbitrarily nested procedure calls.

Lemma 6.8. *Let τ be a WF, non-empty trace with matched* push()/pop() *pairs, whose first state is $\langle \mathtt{skip}, s, h, \_\rangle$ and* last$(\tau) = \langle _, s', h', _\rangle$*. Let π be an access path which is disconnected and acyclic in the well-behaved state s, h, that $\alpha(\{\tau\}) = \langle W, \_, \_\rangle$, and that $\forall \mathsf{e} \in W.\, \mathsf{e} \not\prec \pi$.*
Then, π is preserved from s, h to s, h'.

Proof. By induction over subtraces corresponding to method calls, using Lemma 6.7. □

While we have shown that a stable path is preserved along a trace, this only applies to balanced pairs of push()/pop() commands in the trace. However, the racy access may occur inside a chain of procedure calls, thus we need to further show that a stable path propagated through the call stack is also preserved, up to the point of access.

Lemma 6.9. *Let τ be a (prefix) trace produced by Lemma 5.6. If additionally it is the case that $\forall \mathsf{e} \in \alpha(\{\tau\})_W.\, \mathsf{e} \not\prec \pi$ then $\lfloor \pi \rfloor_{s,h} = \lfloor \pi' \rfloor_{s',h'}$.*

Proof. We split $\tau = \tau_1 +\!\!+ \tau_{\mathrm{push}} +\!\!+ \tau_0$ on the last unmatched push() command, and apply Lemma 6.8 to τ_0, showing that π' is preserved. Then, we lift $\lfloor \pi' \rfloor_{s',h'}$ to the parent call stack and identify the argument e_j such that $\mathrm{root}(\pi') = \mathtt{arg}_j$. Then, we repeat the process for e_j as π' up to the top-level stack. See the extended version (Gorogiannis et al. 2018) for details of the proof. □

We are ready to state the main result of this section and the whole paper.

Theorem 6.10 (True Positives Theorem). *Let C_1, C_2 be two programs such that $[\![C_i]\!]^\# \bot_{\mathcal{D}} = \langle W_i, \_, A_i \rangle$. Let π_i be two paths and c_i two commands such that $\langle c_i, L_i \rangle \in A_i$, and,*

- *$\pi_1 = v_1.\bar{\mathrm{f}}$ and $\pi_2 = v_2.\bar{\mathrm{f}}$ (i.e., the field sequences are the same);*
- *$\forall \mathrm{e}_i \in W_i.\, \mathrm{e}_i \not\prec \pi_i$ (for $i \in \{1, 2\}$);*
- *$c_1 = (\pi_1 := x)$ and $c_2 = (\pi_2 := y)$, or, $c_1 = (\pi_1 := x)$ and $c_2 = (y := \pi_2)$;*
- *$L_1 + L_2 \leq 1$.*

Then there exists a well-behaved state $\varsigma = \langle (s_1, s_2), h, (0, 0) \rangle$ and a concurrent trace $\tau^{\|} \in [\![C_1 \parallel C_2]\!]\, \varsigma$ that races.

Proof. We sketch the important intuitions behind the proof. For more details, see the extended version (Gorogiannis et al. 2018).

By assumption, one of $L_1 = 0 \vee L_2 = 0$, so *w.l.o.g.*, we set $L_1 = 0$. This means that when C_1 accesses the path π it is *not holding the lock*.

We first construct a special initial state, where both programs "see" the same path on the shared heap, and where that path is acyclic and disconnected.

We then obtain a partial trace τ_1 of C_1 as ordained by Lemma 5.6, that ends just before committing an access to π.

We further show that the final state of that trace satisfies the preconditions of Lemma 5.6 for C_2 thus obtaining a trace τ_2 that picks up from where τ_1 finished.

We weave τ_1 and τ_2 into a concurrent trace by letting C_1 progress until it reaches the end of τ_1, and then C_2 progresses until the end of τ_2. Crucially, this schedule is realisable because of the above observation that C_1 isn't holding the lock.

We finally use Lemmas 6.8–6.9 to show that the path is preserved to the concurrent trace' final state, and that the path each thread sees through its call stack resolves to the same address. □

7 IMPLEMENTATION AND EVALUATION

In the previous sections we laid out the design of a theoretical analyser that is similar to RacerD, but which enjoys the True Positives property: under certain assumptions, reports from the theoretical analyser proposed here, are true positives. But that still does not address the effectiveness of the new analyser: we would like to know would an implementation of RacerDX be

(1) effective, in that it reports a sufficient proportion of reports which RacerD generates;
(2) efficient, in that it is not significantly slower than RacerD.

We implemented the proposed analyser as a modification of RacerD. The implementation was relatively straightforward: it required less than 1kLOC of OCaml code, and used data structures and algorithms that are either standard or come with the Infer analyser.

We next discuss the differences between the two analysers.

7.1 Differences between RacerD and RacerDX

The main difference between RacerD and RacerDX is in substituting the ownership domain of the former with that of stability.

Newly allocated objects in Java are known only to the caller of **new**. Thus, it is not possible for two threads to race on a newly allocated object, unless the address of that object flows into some shared location (basically, the heap). In addition, accesses to fields of the newly allocated object cannot possibly participate in a data race, for much the same reasons. RacerD uses a notion of *ownership* typing to track access paths which have been obtained through allocation, in an effort to avoid the associated false positives.

RacerDX swaps the whole abstract domain of ownership for that of stability. This substitution is logically warranted by the fact that allocation immediately destabilises all paths extending that of

the receiver. For instance, in the code `x.b = new Bloop(); x.b.f = 1;` the access at `x.b.f` will not be reported by RacerDX because `x.b` $\prec$ `x.b.f`, and `x.b` will have been recorded in the W component of the analysis state.

7.2 Differences between Theoretical Analyser and its Implementation

Mathematical formulations of static analysers typically deal with a simplified language and a simplified semantics for reasons of simplicity and clarity. Their implementations, in so far as the target is a real programming language on a real concrete semantics, usually are not one-to-one renditions of the math in code. Here we examine some of the differences between the formal model of RacerDX and its implementation. Some of these differences are directly inherited from RacerD; these include a treatment of pure procedures (aided by programmer annotations); support for confining a set of accesses in a specific thread (for instance, a UI thread) such that no lock is required to avoid interference; **`static`** and **`volatile`** fields; inheritance, and others.

The parts of RacerD we had to specifically modify for stability, and which constitute a complex feature not present in the formal model of RacerDX include:

Global variables: These clearly do not fit the model of RacerDX, as they do not behave as formals. We model them in the implementation as paths that never undergo substitution, and are never destabilised. This means that any access to them not under synchronisation may be reported in the context of a race.

Constructors: A constructor cannot normally race on the freshly created object. To avoid the obvious false positives, we treat **`this`** as always unstable when inside a constructor. Thus, **`this`** $\in W$ and therefore any statement that initialises a member field will not be reported because **`this`**$\prec$**`this`**`.f` for any field `f`. Note that this allows potential false negatives in the case where the address of **`this`** (or some path starting at **`this`**) escapes the constructor.

Containers: Such objects are treated specially by RacerD, because they are usually opaque (live in libraries) and because the model they expose is much simpler than their implementation. For instance, an addition of an element to an `ArrayList` at the path **`this`**`.arrayList` will be treated as a store, for the purposes of thread safety. However, RacerDX needs to distinguish between accesses to the container itself and accesses that manipulate the contents, for instance **`this`**`.arrayList = f()` and **`this`**`.arrayList.get(0)`. The way this is achieved is through a dummy field that is read/written to when an element is read/added from/to the container.

7.2.1 Analysing Open-Source Projects. We obtained six open-source Java projects (*cf.* Table 1a) and analysed them with RacerD and RacerDX, recording their total CPU time cost and data race reports. The test environment was Linux 4.11, running on a server with 56Gb of RAM, on an Intel CPU at 2.5GHz. Results are shown in Table 1b. The total CPU time of each tool (in seconds) and the proportion of their difference are given in the columns "D CPU", "DX CPU", "CPU ±%".

An access path may be accessed at a set of locations. In the worst case where are these accesses are racy, we may end up with a number of reports that is quadratic in the number of locations. RacerD and RacerDX both have de-duplication capacilities to avoid spamming developers, thus selecting a subset of races. Since, for the purposes of the evaluation, this selection is arbitrary, we deactivated de-duplication in both tools to ease the comparison. We checked that RacerDX never introduces reports, that is, if RacerDX makes a particular report then RacerD always makes the same report too. We show the number of race reports for both tools and the proportion of the difference in the columns "D Reps", "DX Reps" and "Reps ±%".

To further elucidate the fundamental differences, we extracted the access path π a race report identifies, and counted how many unique paths are reported by each tool. Again, we found that

Table 1. Evaluation: target projects and statistics.

Project	Description	URL
avrora	An AVR emulator	https://github.com/ibr-cm/avrora
Chronicle-Map	A non-blocking key-value store	https://github.com/OpenHFT/Chronicle-Map
jvm-tools	Tool for JVM troubleshooting and profiling	https://github.com/aragozin/jvm-tools
RxJava	Library for asynchronous and event-based programs	https://github.com/ReactiveX/RxJava
sunflow	Rendering system for photo-realistic image synthesis	https://github.com/fpsunflower/sunflow
xalan-j	XSLT processor	https://github.com/apache/xalan-j

(a) Evaluation targets.

Target	LOC	D CPU	DX CPU	CPU ±%	D Reps	DX Reps	Reps ±%	D #π	DX #π	#π ±%
avrora	76k	103	102	0.4%	143	92	36%	78	38	51%
Chronicle-Map	45k	196	196	0.1%	2	2	0%	2	2	0%
jvm-tools	33k	106	109	-3.6%	30	26	13%	14	11	21%
RxJava	273k	76	69	9.2%	166	134	19%	65	44	32%
sunflow	25k	44	44	-1.4%	97	42	57%	116	38	67%
xalan-j	175k	144	137	5.0%	326	295	10%	135	94	30%

(b) Evaluation results. CPU columns are in seconds; Reps are distinct reports; π are distinct paths.

RacerDX never reports a path not also reported by RacerD. We show the number of paths reported in the columns "D #π", "DX #π" and the proportion of the difference in "#π ±%".

We can make the following observations:

- The difference between runtimes is largely within the noise margins, especially given that a large percentage of these runtimes is spent compiling Java source into bytecode, as Infer extracts an AST from the compiled artefact.
- The loss in terms of number of reports ranges between 10% and 57% (we exclude Chronicle-Map as there are too few reports to start with), and the loss in terms of number of distinct access paths ranges from 21% to 67%.

7.2.2 The Causes for Deterioration of Reporting Rate. We triaged a sample of reports that RacerD made but RacerDX didn't. We discerned two main classes of reports:

- In a call `this.foo(this.f)`, the check whether one of the actuals is a proper prefix of another (Figure 7, method call case) fails because `this`$<$`this.f`. Thus, `this` is marked as unstable (is added to the W component of the abstract state). But this means that all accesses in the caller method will not be considered in data race reports, leading to potential false negatives.
  A potential solution here may be to use the fact that the first argument of a non-`static` Java method cannot be reassigned, and thus may be left out of the check above, but we have not at present assessed how this might affect the status of our theorems.
- Inner (nested) classes are common in Java, and allow methods of an inner class object to reference fields and methods of the containing class. To achieve this, the compiler inserts in the inner class a hidden reference to the outer class object, and initialises this appropriately at construction. Unfortunately, this also means that the `this` reference of the outer class is marked as unstable whenever an inner class object is constructed, thus precluding accesses occurring in the enclosing method from being reported.

Remarkably, all the reports we triaged were true positives. Both of these classes of missing reports may benefit from elaborating the stability abstract domain to track escaping references, *i.e.*, when a path is read it is not immediately marked as unstable, but only when the address read ends up being stored somewhere. This is something we will investigate in further work.

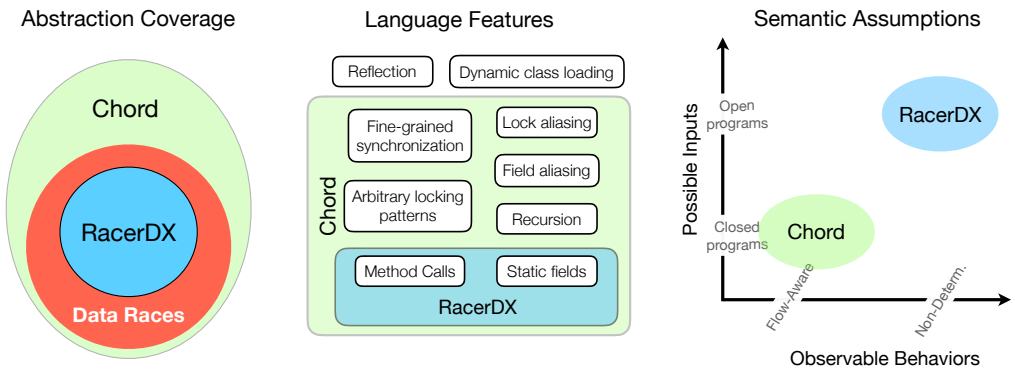

Fig. 9. The aspects of a race analyzer: abstraction coverage, language features, and semantic assumptions.

8 DISCUSSION AND RELATED WORK

In this section we place RacerDX in the field of static race detectors that have been subjected to formal analysis. Since the majority of the static race analyzers used in industry—ThreadSafe (Atkey and Sannella 2015), Coverity's analysis suite (Chou 2014), and RacerD (Blackshear et al. 2018)—despite being impactful in practice, do not come with any formal theorem about their algorithm, we focus on the only two race analyzers we know of, for which a formal statement has been made: Chord (Naik et al. 2006; Raghothaman et al. 2018) and RacerDX.

Chord is a tool which strives to be *sound*, or to favour reduction of false negatives over false positives. Note that our discussion should not be interpreted as providing value judgements on RacerDX versus Chord. We think of both as exemplars of points in the design space which might be built upon or learned from in the construction of new analyzers that are useful in practice.

The notion of soundness is a standard one: the results of an analysis model *all possible executions* of the program *wrt.* certain behaviors of interest, *e.g.*, uncaught exceptions, memory leaks, *etc.* This is the concept of over-approximation. A consequence of an over-approximation result is that if an analyzer claims that there are no bugs then there are none (at least in the idealized formal model regarded as the concrete semantics).

RacerDX, on the other hand, is an example of a static analysis that favours reduction of false positives over false negatives. It aims to be *complete*, to report only true races, under certain assumptions. A series of diagrams, shown in Figure 9, illustrate specific aspects of the analysis design space and depict the notions of *over-of-under* and *under-of-over* approximation from the introduction. We discuss each of the diagrams.

- *Abstraction Coverage.* The left part of Figure 9 demonstrates the relative positioning of claims by Chord and RacerDX. Chord strives to compute an over-approximation, so that its data race reports should be a superset of the races than can occur in program executions. In contrast, RacerDX targets *under-approximation*, computing a subset of the real races, which is what our True Positives Theorem addresses.
- *Language Features.* Claims about *unconditional* absence of false negatives or false positives made with regard to an industrial-scale project and a realistic programming language (*e.g.*, Java), are often false for particular analyzers, even if their designs are guided by considerations of soundness or completeness. The set of choices of which language features to model faithfully or to ignore, which we refer to as *context*, affects both the soundness and completeness claims, and frame the analysis' claims, serving as signs of warning to the analysis' users. The *"Soundiness"*

movement (Livshits et al. 2015) mandates the designers of new static analyses to make the context explicit and offers a list of likely causes of unsoundness in popular languages and runtimes; that is, soundness (over-approximation) can be wrt a restriction (under-approximation) of the chosen concrete semantics. Similarly, for analyzers aiming at completeness the context should be made explicit.[11] As an example, the central part of Figure 9 shows a selection of features, framing the claims made for Chord (with extensions to its points-to analysis component by Naik and Aiken (2007)) and RacerDX, correspondingly. In the latter case, it corresponds to the assumptions **A1**–**A5** we made in Section 3.

- *Semantic Assumptions.* The main reasons for having the claims framed properly with regard to the context, is to make sure that the class of the programs satisfying the context assumptions, can be faithfully represented by a semantics, with respect to which the soundness or completeness claims were established. But the choice of the semantics itself, as well as non-language-specific assumptions, is rarely questioned, or exercised to the advantage of the analysis designer. For instance, a soundness result for a data race analysis, proven modulo a rigorously stated context, can be invalidated if, for instance, one assumes a version of a relaxed-memory semantics (Kang et al. 2017) instead of more traditional sequential consistency (Lamport 1979). At the same time, a completness result would be *simpler* to establish in this setting, as it would allow one to *choose from more program behaviors*! The right part of Figure 9 depicts this remarkable variance of the soundness/completeness results with regard to the chosen foundations, determining the set of allowed behaviors and inputs.
  An analysis aiming for soundness will be easier to prove in a more restrictive semantics and in closed-world assumptions, which is the case for Chord. In contrast, a formal result for an analysis aiming for completeness (*e.g.*, RacerDX) will be easier (or even possible) to state in a less restrictive, possibly non-deterministic semantics (such as what we assumed via **A6**), in the presence of an open world assumption (which we exploited in proofs in Sections 5 and 6).
  As far as we are aware, this observation is novel, although in retrospect not very surprising.

In the framework of Figure 9, our True Positives Theorem delivers *wrt.* claims on relating to the set of unwelcome behaviors (No-False-Positives), framed within a particular context *wrt.* supported language features (assumptions **A1**–**A5**) for carefully chosen relaxed semantic assumptions **A6**.

Over-of-Under and Under-of-Over. In the light of the suggested classification the over-of-under and under-of-over characterization of an analyzer is a matter of positioning the *claims* and choosing the *foundations*. Analyzers claiming a version of soundness to prevent bugs, such as Chord, typically establish this in somewhat restricted foundations (*e.g.*, sequentially-consistent semantics with no non-determinism, closed-world assumptions), but over-approximates the set of unwelcome behaviors, therefore delivering the over-of-under result. Conversely, analyzers aiming for completeness to detect bugs, such as RacerDX, choose to consider a more relaxed semantics, while under-approximating the set of the bugs that they can report, going only for high-confidence ones, which corresponds to the under-of-over result.

Regardless of whether the over-of-under or under-of-over approach is chosen, how can one rigorously validate the set of choices made with regard to the supported features and semantic assumptions? In the case of soundness claims, one can instrument code with checks that reflect the sources of deliberate unsoundness (Christakis et al. 2015) and compute the fraction of the code that is analyzed soundly. Another way, which, in our experience, worked well for completeness/TP

[11] However, we avoid inventing a new term, as both soundiness and completeness-relative-to-assumptions can, as far as we are aware, be formulated as standard soundness or completeness results wrt models capturing the assumptions (which themselves can often be under-approximate or over-approximate models).

claims, is to *instrument the analysis itself*, so that it would not report bugs that it is not absolutely adamant about—precisely what is achieved by our notion of wobbliness/stability from Section 4.[12]

Relevance of Theory. How can we establish that the formal analysis of an analyzer is relevant in practice, especially when the contextual assumptions for its theorems might be violated?

For analyzers that come with a proof of soundness, the criterion often used is that of *precision*, *i.e.*, that the analyzer does not report too many false positives. The rate of true positives can be improved by chaining the main analysis with a suitably tailored *pre-analysis* (Oh et al. 2014), or by involving a user into the loop and taking her feedback into account (Raghothaman et al. 2018).

Our approach, which addresses this challenge for an completeness-style analyzer, is a complementary one and assesses whether the analysis's *generality* does not deteriorate (*i.e.*, whether it does not ignore too many plausible bugs). To do this, we conducted an empirical comparison to an already deployed tool, which has been considered highly impactful by industry standards (Blackshear et al. 2018), and demonstrate that a rigorously proven version does not do too much worse (Section 7).

8.1 Theorems for Completeness-oriented Race Detectors

Several results on proven *dynamic* concurrency analyses have been published recently. The pioneering work by Sadowski et al. (2008) described a partial verification of some properties of the Velodrome (Flanagan et al. 2008) dynamic atomicity checker, but considered only a model of the algorithm, not its actual implementation. The work by Mansky et al. (2017) formalized the notion of run-time data races on memory locations and proved the soundness and completeness of the specification rules for a number of idealized implementations of race detection analyses, including FastTrack (Flanagan and Freund 2009). Finally, Wilcox et al. (2018) verified a more realistic implementation of FastTrack, showing that it is as fast as or faster than existing state-of-the-art non-verified FastTrack implementation.

The paper of Blackshear et al. (2018) compares RacerD to several dynamic analyses. We are not making claims here about the relative usefulness of the dynamic and static techniques, except to say that they have often been found to be complementary.

To the best of our knowledge, our result is the first formal proof of a No-False-Positives property for a *static* concurrency analysis.

8.2 Proving Static Analyzers Complete

The subject of completeness is well-studied in the Abstract Interpretation community in the context of over-approximating (soundness) analyzers. For instance, the work by Ranzato (2013) demonstrates how completeness can be crucial for designing static analyzers for a number of common intra-procedural properties (*e.g.*, signs, constant propagation, polyhedra domains, *etc*), encouraging one to reason about the completeness properties of their underlying abstract domains. Giacobazzi et al. (2015) go even further, providing a *proof system* for showing that the result of a certain analysis on a particular given program is precise. Those works address mostly numerical properties. As far as we know, our work is the first to address completeness of an abstraction and a single-threaded abstract semantics for detection of concurrent races.

9 CONCLUSIONS

In this paper we have formulated and proven a True Positives Theorem for an idealized static race detector, motivated by one that has been proven to be effective in production at Facebook. We have also provided an empirical evaluation of the distance between the idealized analyzer and the in-production version.

[12] That said, wobbliness does not account for all of our language features, such as, *e.g.*, well-scoped locking (**A2**).

It is important to emphasize that we don't view the primary role of the True Positives Theorem as providing guarantees to programmers. Rather, it clarifies to the analysis designer the nature of the analysis algorithm: its purpose is to provide understanding and guide design of current and future analyzers. This kind of role of theory is complementary to the goal of confirming properties of analyses after they have been finished, and is a relative of the corresponding role of semantics in programming language design advocated by Tennent (1977). Indeed, the theorem was not formulated before Blackshear et al. (2018) implemented RacerD, as a specification for them to meet. Rather, the theorem arose in response to the fact that the tool did not satisfy a standard over-approximation (soundness for bug prevention) or under-approximation (completeness, or soundness for testing) result, and yet was effective in practice. And, as we mentioned, the statement of the theorem then impacted the further development of RacerD.

We have not formulated a general theory for the TP theorem, into which many analyzers would fit. The reason is not that we can't see how to make *some* such theory, but rather that we are being careful not to engage in quick generalisation from few examples from practice. Beyond the *shape* of the TP theorem (over-of-under), the more significant issue of the nature of the assumptions supporting a meaningful theorem is one that we would like to see validated by other in-production analyses, before generalization. Formal theorems about static analyses for data race detection have concentrated on soundness results, where here we considered completeness. In both cases contextual assumptions often need to be stated which concern, *e.g.*, language features treated and ignored or other assumptions reflected in the abstractions.

The way we compose over- and under-approximations is consistent with the spirit of abstract interpretation. Although abstract interpretation can be used to formulate and prove vanilla soundness and completeness results (over and under on their own), perhaps its greater value comes from composing and comparing different abstractions, to give finer insight on the nature of analyses than the vanilla results (which often do not literally hold) would do. It is obvious that one *might* consider any sequence of over- and under-approximations. A modest suggestion of this work is that perhaps under-of-over is deserving of more attention, as a way to study static analyses that are designed for bug catching rather than prevention.[13]

ACKNOWLEDGMENTS

First and foremost, we thank Sam Blackshear for his amazing work on RacerD implementation. We are grateful to Don Stewart for his encouragement and support for this research project. We thank Francesco Logozzo for his comments on the formalisation of the analysis completeness. Finally, we wish to thank the POPL'19 PC and AEC reviewers for the careful reading of the paper, and for many insightful comments and suggestions.

REFERENCES

Robert Atkey and Donald Sannella. 2015. ThreadSafe: Static Analysis for Java Concurrency. *ECEASST* 72 (2015).

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *PACMPL* 2, OOPSLA (2018), 144:1–144:28.

Stephen Brookes. 2007. A semantics for concurrent separation logic. *Th. Comp. Sci.* 375, 1-3 (2007).

Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

Andy Chou. 2014. From the Trenches: Static Analysis in Industry. (2014). Invited keynote talk at POPL'14. Available at https://popl.mpi-sws.org/2014/andy.pdf.

Maria Christakis, Peter Müller, and Valentin Wüstholz. 2015. An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer. In *VMCAI (LNCS)*, Vol. 8931. Springer, 336–354.

[13] This suggestion is consistent with bounded model checking and symbolic execution for testing, which can often be seen as computing *either* over-of-under or under-of-over.

David G. Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*. ACM, 292–310.

Patrick Cousot. 1978. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* Ph.D. Dissertation. Université Scientifique et Médicale de Grenoble.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM Press, 269–282.

Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *J. Log. Comput.* 2, 4 (1992), 511–547.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *PLDI*. ACM, 121–133.

Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*. ACM, 293–303.

Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *POPL*. ACM, 261–273.

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice.* Addison-Wesley.

Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. A True Positives Theorem for a Static Race Detector – Extended Version. *CoRR* 1811.03503 (2018). arXiv:1811.03503 https://arxiv.org/abs/1811.03503

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming.* M. Kaufmann.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. ACM, 175–189.

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

William Mansky, Yuanfeng Peng, Steve Zdancewic, and Joseph Devietti. 2017. Verifying dynamic race detection. In *CPP*. ACM, 151–163.

Mayur Naik and Alex Aiken. 2007. Conditional must not aliasing for static race detection. In *POPL*. ACM, 327–338.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *PLDI*. ACM, 308–319.

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*. ACM, 475–484.

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. Interactive Program Reasoning using Bayesian Inference. In *PLDI*. ACM, 722–735.

Francesco Ranzato. 2013. Complete Abstractions Everywhere. In *VMCAI (LNCS)*, Vol. 7737. Springer, 15–26.

Caitlin Sadowski, Jaeheon Yi, Kenneth Knowles, and Cormac Flanagan. 2008. Proving correctness of a dynamic atomicity analysis in Coq. In *Workshop on Mechanizing Metatheory*.

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. *Proceedings of the Workshop on Binary Instrumentation and Applications*, 62–71.

Robert D. Tennent. 1977. Language Design Methods Based on Semantic Principles. *Acta Inf.* 8 (1977), 97–112.

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *POPL*. ACM, 343–356.

James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. 2018. VerifiedFT: a verified, high-performance precise dynamic race detector. ACM, 354–367.

Hongseok Yang and Peter O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, 402–416.